4

RADC-TR-88-324, Vol IX (of nine)
Interim Report
March 1989

NORTHEAST ARTIFICIAL INTELLIGENCE CONSORTIUM ANNUAL REPORT 1987 Computer Architectures for Very Large Knowledge Bases

Syracuse University

P. Bruce Berra, et al

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-88-324, Vol IX (of nine) has been reviewed and is approved for publication.

APPROVED:

RAYMOND A. LIUZZI
Project Engineer

APPROVED:

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: John A. Ritz

JOHN A. RITZ
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTC) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

1b. RESTRICTIVE MARKINGS
N/A

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE
N/A

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release; distribution unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
N/A

5. MONITORING ORGANIZATION REPORT NUMBER(S)
RADC-TR-88-324, Vol IX (of nine)

6a. NAME OF PERFORMING ORGANIZATION
Northeast Artificial Intelligence Consortium (NAIC)

6b. OFFICE SYMBOL *(If applicable)*

7a. NAME OF MONITORING ORGANIZATION
Rome Air Development Center (COTC)

6c. ADDRESS *(City, State, and ZIP Code)*
409 Link Hall
Syracuse University
Syracuse NY 13244-1240

7b. ADDRESS *(City, State, and ZIP Code)*
Griffiss AFB NY 13441-5700

8a. NAME OF FUNDING / SPONSORING ORGANIZATION
Rome Air Development Center

8b. OFFICE SYMBOL *(If applicable)*
COES

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F30602-85-C-0008

8c. ADDRESS *(City, State, and ZIP Code)*
Griffiss AFB NY 13441-5700

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.
61102F	2304	J5	01
62702F	5581	27	13

11. TITLE *(Include Security Classification)*
NORTHEAST ARTIFICIAL INTELLIGENCE CONSORTIUM ANNUAL REPORT 1987 Computer Architectures for Very Large Knowledge Bases

12. PERSONAL AUTHOR(S)
P. Bruce Berra, et al

13a. TYPE OF REPORT
Interim

13b. TIME COVERED
FROM Dec 86 TO Dec 87

14. DATE OF REPORT *(Year, Month, Day)*
March 1989

15. PAGE COUNT
220

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
12	05	

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Very Large Knowledge Bases, Artificial Intelligence, Computer Architectures, Real-Time Processing, Pattern Matching, Parallel Computing

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

The Northeast Artificial Intelligence Consortium (NAIC) was created by the Air Force Systems Command, Rome Air Development Center, and the Office of Scientific Research. Its purpose is to conduct pertinent research in artificial intelligence and to perform activities ancillary to this research. This report describes progress that has been made ~~in the third year of~~ the existence of the NAIC on the technical research tasks undertaken at the member universities. The topics covered in general are: versatile expert system for equipment maintenance, distributed AI for communications system control, automatic photo interpretation, time-oriented problem solving, speech understanding systems, knowledge base maintenance, hardware architectures for very large systems, knowledge-based reasoning and planning, and a knowledge acquisition, assistance, and explanation system. The specific topic for this volume is the development of architectures for very large knowledge bases, especially in light of real-time requests, parallelism, and the advent of optical computing.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL
RAYMOND A. LIUZZI

22b. TELEPHONE *(Include Area Code)*
(315) 330-2925

22c. OFFICE SYMBOL
RADC/COTC

Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Table of Contents

9. Computer Architecture for Very Large Knowledge Bases

9.1. Executive Summary

The focus of our research is on the development of algorithmic, software and hardware solutions for the management of very large knowledge bases (VLKB) in a real time environment. We assume a logic programming inferencing mechanism and a relational model for the management of the knowledge base. The interface between the inferencing mechanism and the extensional data base becomes one of partial match retrieval. During 1987 we have conducted research on many aspects of this problem as indicated in this report.

We completed the analysis and simulation of surrogate file structures. We considered concatenated code words (CCW), superimposed code words (SCW) and transformed inverted lists (TIL). Our primary technique will be CCW but we will also utilize SCW and TIL in some our research. In addition to good overall performance CCW offer some interesting additional attributes. Namely, relational operations can be performed directly on the surrogate file when it is structured using CCW. The further development of this has become the doctoral dissertation topic of one of the students on the program. This work has a direct effect on the set of operations each surrogate file processor will be required to perform and therefore on its design.

We have begun working on a demonstration system that will be used to interface with a logic programming language, generate surrogate files and manage a knowledge base. The system consists of Prolog, specially developed modules and the INGRES data base management system.

We have extended the TIL concept to the management of very large dynamic knowledge bases. This has led to the development of a new access method which we call the dynamic random-sequential access method (DRSAM). This has become the doctoral dissertation topic of one of the students on the project.

In a very large knowledge base the number of rules may be so large that special architectures may be required for the management of the rules. We have investigated the use of CCW for the management of the rules and have developed an associative memory approach. The approach is based on guarded horn clauses and mode declarations in a parallel logic programming context.

Another area that we have been investigating is the potential role of optical storage, interconnection and processing in the management of VLKB. We have developed approaches to the processing of digital light signals coming from optical storage media via optical interconnection. The division between types of processes is between those that do not require intermediate storage and those that do. The performance of a robust set of relational operations through optical means has become the doctoral dissertation topic of one of the graduate students on the project.

Finally, all of the above work supports the long range development of a VLKB architecture. During this year we were able to provide considerably more detail regarding the specifications for that system.

9.2. Introduction

Knowledge based systems consist of rules, facts and an inference mechanism that can be utilized to respond to queries posed by users. The objective of such systems is to capture the knowledge of experts in particular fields and make it generally available to nonexpert users. The current state of the art of such systems is that they focus on narrow domains, have small knowledge bases and are thus limited in their application.

As these systems grow, increased demands will be placed on the management of their knowledge bases. The intensional database (IDB) of rules will become large and present a formidable management task in itself. But, the major management activity will be in the access, update and control of the extensional database (EDB) of facts because the EDB is likely to be much larger than the IDB. The volume of facts is expected to be in the gigabyte range, and we can expect to have general EDB's that serve multiple inference mechanisms. In this report we assume that the IDB is a set of rules expressed as logic programming clauses and the EDB is a relational database of facts.

In order to set the stage for the problem that we are interested in, consider the following simple logic programming problem:

```
1. grandfather(X,Y) ← father(X,Z), parent(Z,Y)
2. parent(X,Y) ← father(X,Y)
3. parent(X,Y) ← mother(X,Y)
4. father(pat, tiffany) ←
   father(don, louise) ←
        .
        .
        .

5. mother(mary, louise) ←
   mother(lisa, tiffany) ←
        .
        .
        .

6.  ← grandfather(X, joan)
```

The first three clauses form the IDB of rules for this problem, the next two sets form the EDB of facts and the last statement is the goal. To solve the problem (satisfy the goal), we must find the names of the grandfathers of joan. For this we search the father and mother facts on the second argument position, finding values for the first argument position that can be used later. Thus, we need to find joan's mother and father before finding her grandfathers. If we ask a similar but slightly different query

```
← grandfather(tom, X)
```

we search the first argument of the father and mother facts in attempting to satisfy it.

Consider the following general goal statement of a logic programming language

$$\leftarrow r\,(X_1,X_2, \cdots ,X_n).$$

In this case, values for some subset of the X_i's will be given in the process of trying to satisfy its goal. Since the subset of the X_i's is not known in advance and can range from one to all of the values, this places considerable requirements on the relational database management system that supports the logic programming language. In fact, in order to insure minimum retrieval time from the relational database all of the X_i's must be indexed. With general indexing the index data could be as large as the actual EDB. In order to considerably reduce the amount of index data yet provide the same capability, we have considered surrogate files. Obviously if not all of the X_i's can take part in goal satisfaction then the indexing strategy will change, however in this report we will assume the most general case in which all of the X_i's are active.

Retrieving the desired rules and facts in this context is an extension of the multiple-key attribute partial match retrieval problem because any subset of argument positions can be specified in a query and matching between terms consisting of variables and functions as well as constants should be tested as a preunification step.

In the context of very large knowledge bases the question arises as to how to obtain the desired rules and facts in the minimum amount of time. Three reasonable choices of indexing schemes to speed up the retrieval are superimposed code words (SCW), concatenated code words (CCW) and transformed inverted lists (TIL)* surrogate file techniques. Surrogate files are constructed by transformed binary codes where the transform is performed by well chosen hashing functions on the original terms. In [BER87a], SCW, CCW and TIL surrogate files were discussed in terms of the structures, updating procedures, performance of relational operations on the surrogate files, and possible architectures to support them. The term "surrogate file" dates back to early work in information retrieval and other terms, such as "signature file" and "descriptor file" have been used to describe similar structures. [FAL84]

An important advantage of surrogate file techniques is that they can be easily extended for the indexing of the rules expressed as Prolog clauses, where the matching between constants, variables, and structured terms is required to test the unifiability. [RAM86], and [WAD87] have extended the SCW structure for the indexing of Prolog clauses and [SHI87] has extended the CCW structure to index the rules and facts in unified manner.

In section 9.3 of this report we consider SCW and CCW for the management of a very large EDB. In section 9.4 we introduce a software system being developed to demostrate the performance of the SCW and CCW surrogate file

* SCW, CCW and TIL will be singular or plural depending upon the context.

techniques. In section 9.5 we consider two forms of TIL for the management of a very large EDB. In section 9.6 we consider the management of the IDB using CCW and present an initial associative memory architecture. In section 9.7 we consider performing relational operations on optical data. In section 9.8 we present an initial design of a very large knowledge base architecture (VLKBA) and discuss some of its components. Finally, we conclude with some comments on the VLKBA and some of its potential uses.

9.3. Surrogate Files with SCW and CCW

In this section we present SCW and CCW surrogate file techniques for extensional database indexing. Notations that are frequently used in this report are shown in Table 9.3.1.

Notations	Meanings
A_r	Number of arguments in a fact
R_q	Average number of arguments specified in a query
GD	Average number of good drops per query
FD	Average number of false drops per query
S_{db}	Size of the extensional database in bytes
N	Number of facts in the extensional database
S	Size of surrogate file in bits
B	Size of a block in bytes
BR	Binary representation
BCW	Binary code word
QT	Query response time
T_{sp}	Surrogate file processing time
T_{dp}	Extensional database processing time
T_{it}	Intersection time
C_i	Value distribution factor, that is, the average number of facts which have the same value in the i-th argument
C_g	Average of value distribution factor (Average redundancy)

Table 9.3.1. Summary of Notations Frequently Used

9.3.1. System Model for SCW and CCW

9.3.1.1. Superimposed Code Word

Let a tuple D contain A_r argument values, $D=\{d_1,d_2, \cdots ,d_{A_r}\}$. Each argument value ($d_i$, $1 \leq i \leq A_r$) can be mapped into a binary representation (BR) by a well chosen hashing function. The BR can be converted to a binary code word (BCW) with pre-defined length and pre-defined weight, by using a pseudo random number generator. The weight of a BCW is the number of 1's in the BCW. The process of generating a BCW from an argument value is well described in [ROB79]. The SCW of a tuple is generated by ORing A_r BCW's obtained from A_r argument values. A unique identifier is then attached to the SCW and the tuple. This unique identifier serves as a link between the two and is used as a pointer to the EDB or can be converted to an actual pointer to the EDB by dynamic hashing schemes such as linear hashing [LAR82].

Suppose we have a fact type called borders which is given as follows:

borders (Country_1, Country_2, Body_of_Water).

For a particular instance

borders (korea, china, yellow sea)

we would first hash the individual values to obtain BR's, then the BR's would be converted into BCW's and the SCW would be formed as follows:

H(korea)	=	100...01	→	000...100
H(china)	=	010...00	→	001...000
H(yellow sea)	=	110...00	→	100...010
				101...110|00...01

with the BCW's logically ORed together. The unique identifier is attached as shown and the vertical line shows the boundary.

The retrieval process with the SCW surrogate file technique is as follows:

1) Given a query, obtain a query code word (QCW) by ORing BCW's corresponding to argument values specified in the query.

2) Obtain a list of unique identifiers to all tuples whose SCW's satisfy
QCW=QCW .AND. SCW
that is, obtain a list of all SCW's that have 1's in the same position as the QCW by sequentially ANDing the QCW with all entries in the SCW file.

3) Retrieve all the tuples pointed to by the unique identifiers obtained in step 2 and discard the tuples not satisfying the query. These are called "false drops". The facts satisfying the query are called "good drops". The false drops are caused by the non-ideal property of hashing functions and the logical ORing of BCW's which make tuples with different argument values have the same SCW.

4) Return the good drops to the user.

9.3.1.2. Concatenated Code Word

The CCW of a tuple is generated by simply concatenating the binary representations (BR's) of all argument values and attaching the unique identifier of the tuple. With the same example used for SCW, the CCW would be formed as

100...01|010...00|110...00|00...01.

The retrieval process with the CCW surrogate file is as follows:

1) Given a query, obtain a query code word (QCW) by concatenating BR's corresponding to argument values specified in the query. The portion of the query code word for argument values which is not specified in the query is filled with don't care symbols.

2) Obtain a list of unique identifiers to all tuples whose CCW's satisfies

$$QCW=CCW$$

by sequentially comparing the QCW with all CCW's in the CCW file. Note in this case the matching is performed on both 1's and 0's.

3) Retrieve all tuples pointed to by the unique identifiers obtained in step 2 and compare the corresponding argument values of the tuples with the query values to discard the false drops caused by the non-ideal property of hashing functions.

4) Return the good drops to the user.

9.3.2. Simulation and Analysis for SCW and CCW Techniques

Simulations are performed with the equations developed in [CHU87] for the size of surrogate files and the query response time using SCW and CCW techniques assuming that the surrogate files are consecutively stored in a disk, the EDB are randomly stored in a number of disks and the storage utilization of the surrogate file and the EDB is 1. We also assumed that sufficient buffers are available for overlapped operations of block searching and block accessing.

9.3.2.1. Surrogate File Size

For the simulation of the surrogate file size, it is assumed that the EDB remains at the same size regardless of variation of the number of arguments in a tuple (A_r) and 15 bytes are used for each argument value. Therefore, N, the number of tuples in the EDB, can be calculated as follows:

$$N = \left\lfloor \frac{S_{db}}{15 \times A_r} \right\rfloor$$

where S_{db} represents the actual EDB size in bytes not including the unique identifiers for each tuple of the EDB. We also assumed that each argument of a tuple in the EDB has the same redundancy value, C_g, which is the average of the value distribution factors (C_i's) denoting the average number of tuples which have the same value in the i-th argument positions.

$$C_g = \frac{\sum_{i \in A_r} C_i}{A_r}.$$

The results for the size simulation are shown in Figures 9.3.1 through 9.3.2. In Figure 9.3.1 we plot the size of the SCW surrogate file (S_{scw}) as a function of the number of arguments in a tuple (A_r). The size of the surrogate file is expressed as a percentage of the EDB. The EDB sizes are 10^5, 10^7 , and 10^9 bytes while the average number of arguments specified in a query (R_q) takes on the values one and two. Note that S_{scw} increases with the size of the EDB (S_{db}) but decreases with R_q.

In SCW case, if we allow more false drops then the length of the SCW becomes shorter which results in a smaller S_{scw}. However, more false drops leads to more EDB accesses.

In designing the SCW surrogate file one must set the expected number of arguments in a query. In terms of size, the worst case of course is when R_q is 1 and as the value for R_q is set at progressively higher values S_{scw} becomes very small. However, if we assume large R_q in designing the SCW file, we have to allow more false drops than the expected number of false drops, FD, whenever the number of arguments specified in a query is smaller than R_q [ROB79].

In Figure 9.3.2 we plot the size of the CCW surrogate file(S_{ccw}) as a function of the average redundancy(C_g) in the data. Note that with greater redundancy S_{ccw} becomes smaller bec.,use a smaller number of bits can be used for each binary representation. Also note that S_{db} and A_r have significant effects on S_{ccw}.

With regard to the size of surrogate files, we can say that the CCW file technique is better than the SCW technique, even though S_{scw} may be smaller than S_{ccw} when R_q is large, because we assumed that the average number of arguments specified in a query is usually not more than 2. However, in both cases the surrogate file is generally less than 20% of the size of the EDB.

When the size of the EDB is less than 10^7 bytes, the surrogate file size is less than 2 Mbytes, so the whole surrogate file can be stored in a fast memory to speed up the retrieval process.

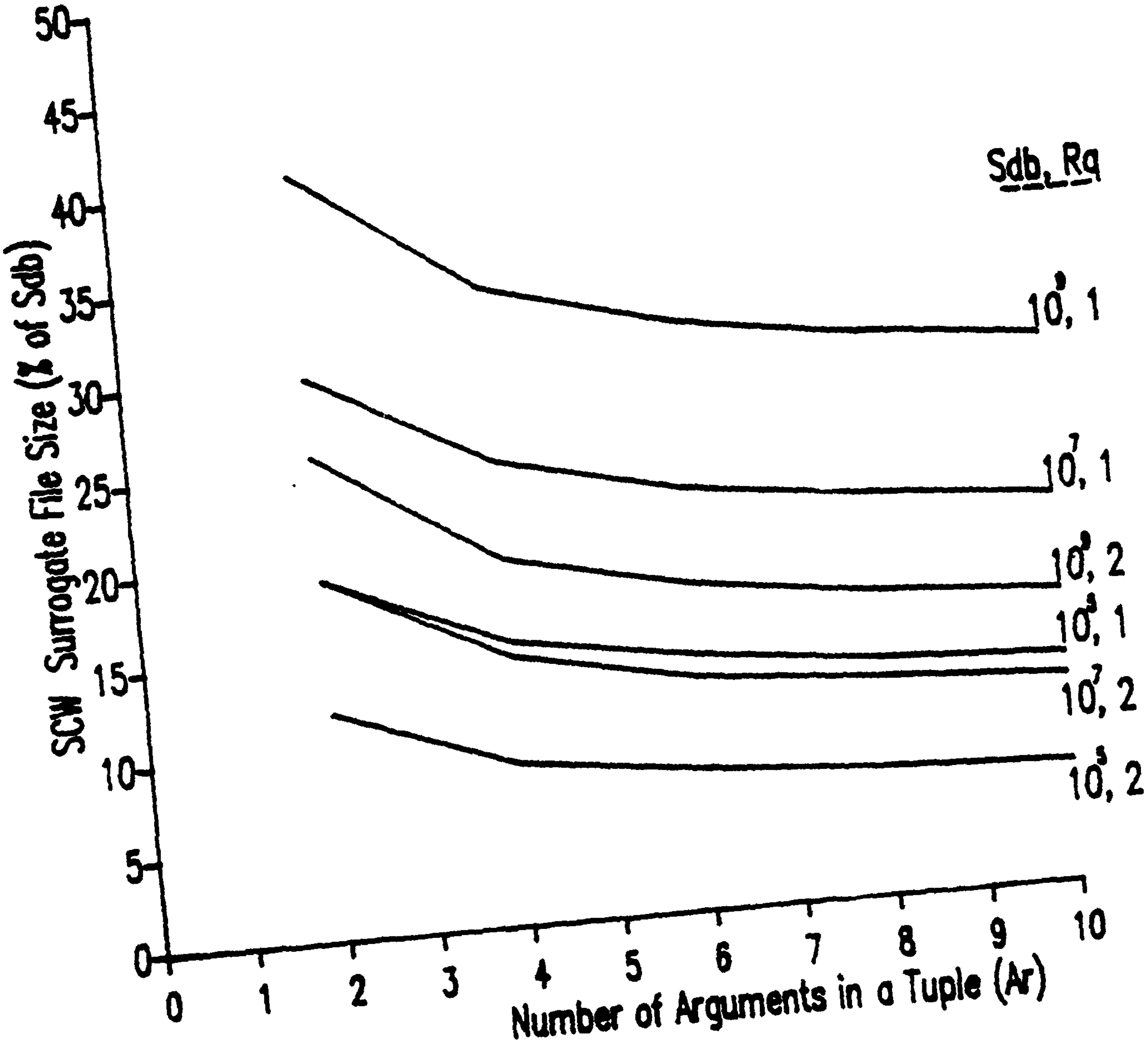


Figure 9.3.1 Effect of EDB Size and the Average Number of Arguments Specified in a Query on the SCW Surrogate File Size (FD= 1)

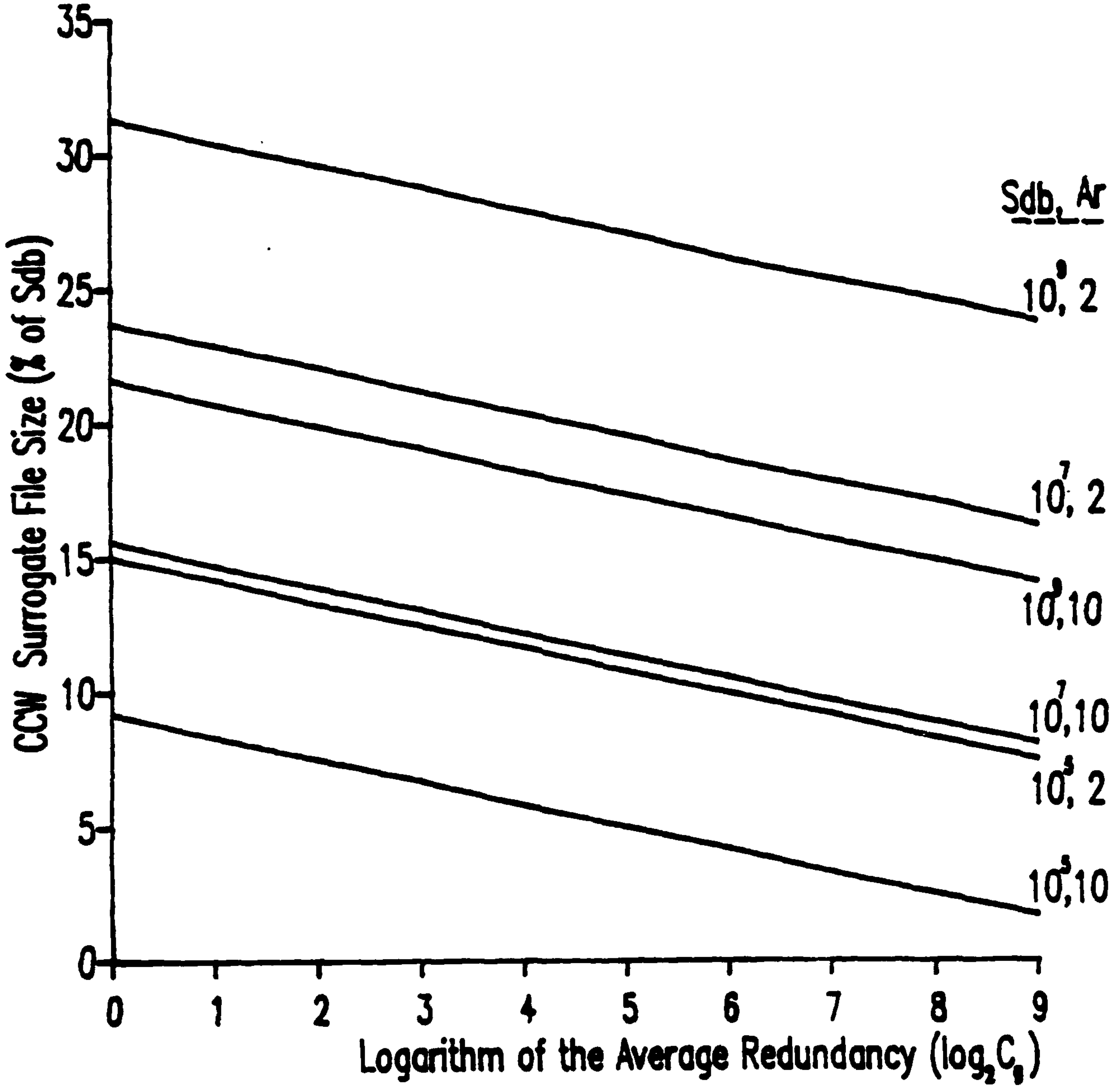


Figure 9.3.2 Effect of Average Redundancy on the CCW Surrogate File Size

9.3.2.2. Query Response Time

For the query response time, we assumed that the hashing function is ideal, so there are no false drops with the CCW surrogate file technique and the SCW surrogate file technique has only the false drops caused by the logical OR operation on the BCW's. A partial-match query is assumed and the BCW of the surrogate file is compared with the QCW by using sequential byte by byte comparison. The query response time results for the SCW and CCW techniques are shown in Figures 9.3.3 through 9.3.6. Table 9.3.2 shows the values of parameters used in this simulation. The parameters relating to the disk are obtained from the characteristics of the DEC RA81 disk [DIG82].

Parameter	Value
Average seek time	28 msec
Minimum seek time	6 msec
Rotational delay	8.3 msec
Data transfer rate	2K bytes/msec
Data sector size	512 bytes
Sectors/track	52
Tracks/cylinder	7
Time for byte comparison	3 μsec
Block size	2K bytes

Table 9.3.2. The Values of Parameters Used in the Simulation

In Figures 9.3.3 through 9.3.4 and 9.3.5 through 9.3.6, we plot the query response times with SCW and CCW surrogate file techniques (QT_{scw} and QT_{ccw}) and corresponding subprocessing times; surrogate file processing time (T_{sp}) and EDB processing time (T_{dp}) for S_{db} of 10^5 and 10^9 bytes, respectively. When S_{db} is 10^5 bytes, most of the query response time is spent for EDB access. But when S_{db} is 10^9 bytes, the query response time becomes very large and most of the query response time is spent for surrogate file accessing and searching because of the increased surrogate file size and sequential searching of the surrogate file. The number of arguments in a tuple (A_r) has little effect on either QT_{scw} or QT_{ccw} since we assumed that the S_{db} remains constant under the variations in A_r.

When S_{db} is 10^5 bytes, R_q is not a factor which affects QT_{scw}, but QT_{scw} increases as FD increases. However, when S_{db} is 10^9 bytes, the result is reversed, that is, R_q affects the QT_{scw} considerably while FD does not. There are two reasons supporting this result:

1) S_{scw} decreases as R_q increases. However, when S_{db} is small, S_{scw} is also small for any R_q so that the time for accessing and searching the SCW file is almost constant. Therefore, the time for accessing the EDB, which depends on FD, becomes a major factor in QT_{scw}.

2) When S_{db} is large, S_{scw} becomes large so that most of QT_{scw} is spent for accessing and searching the SCW file. Therefore, S_{scw} is a main factor deciding QT_{scw}. Since S_{scw} largely depends on R_q, the change in R_q is directly reflected in QT_{scw}.

QT_{scw} and QT_{ccw} are largely affected by C_g when S_{db} is 10^5 bytes and R_q is small. However, as R_q becomes large, the effect of C_g on QT_{scw} and QT_{ccw} decreases. This fact is well explained by the role of R_q and C_g in determining the number of good drops:

1) If R_q is small and C_g is large, then there are so many good drops that a large amount of time is required for accessing the EDB.

2) If R_q becomes large, the number of good drops decrease considerably, and so does the EDB access time, which is the major component of the query response time when S_{db} is 10^5 bytes.

From Figures 9.3.4 and 9.3.6, we can see that when S_{db} is 10^9 bytes, as C_g increases, QT_{scw} remains constant while QT_{ccw} decreases. This occurs because a fewer number of bits is required to uniquely identify each attribute value in the CCW case. But when C_g is larger than a certain value, the query response time starts increasing because of the increased EDB access time. Also, we can see from Figures 9.3.4 and 9.3.6 that most of the query response time is used for the surrogate file accessing and searching when the EDB is large. Therefore, if we use multiple processors and/or associative memory to speed up the surrogate file processing, we can reduce the query response time considerably. Since the surrogate files are quite regular and compact, they can be mapped into the associative memory. Thus, we can obtain a speed up by the content addressing capability and the parallelism of the associative memory [AHU80][BER87]. In addition, we can also obtain a speed up proportional to the number of processors because there is little need for communication among the processors.

Since searching and disk access can be overlapped, if we increase the block size, then the number of disk accesses can be reduced and we can save time as long as the block searching time is less than the block access time. In the case of a multiple disk system, the surrogate file and the EDB are distributed over a number of disks and we can reduce the disk access time by seeking several disks concurrently.

Comparing the retrieval performance of the SCW and CCW techniques, we can see that QT_{ccw} is smaller than QT_{scw} when R_q is small, because S_{ccw} is smaller than S_{scw} when R_q is small.

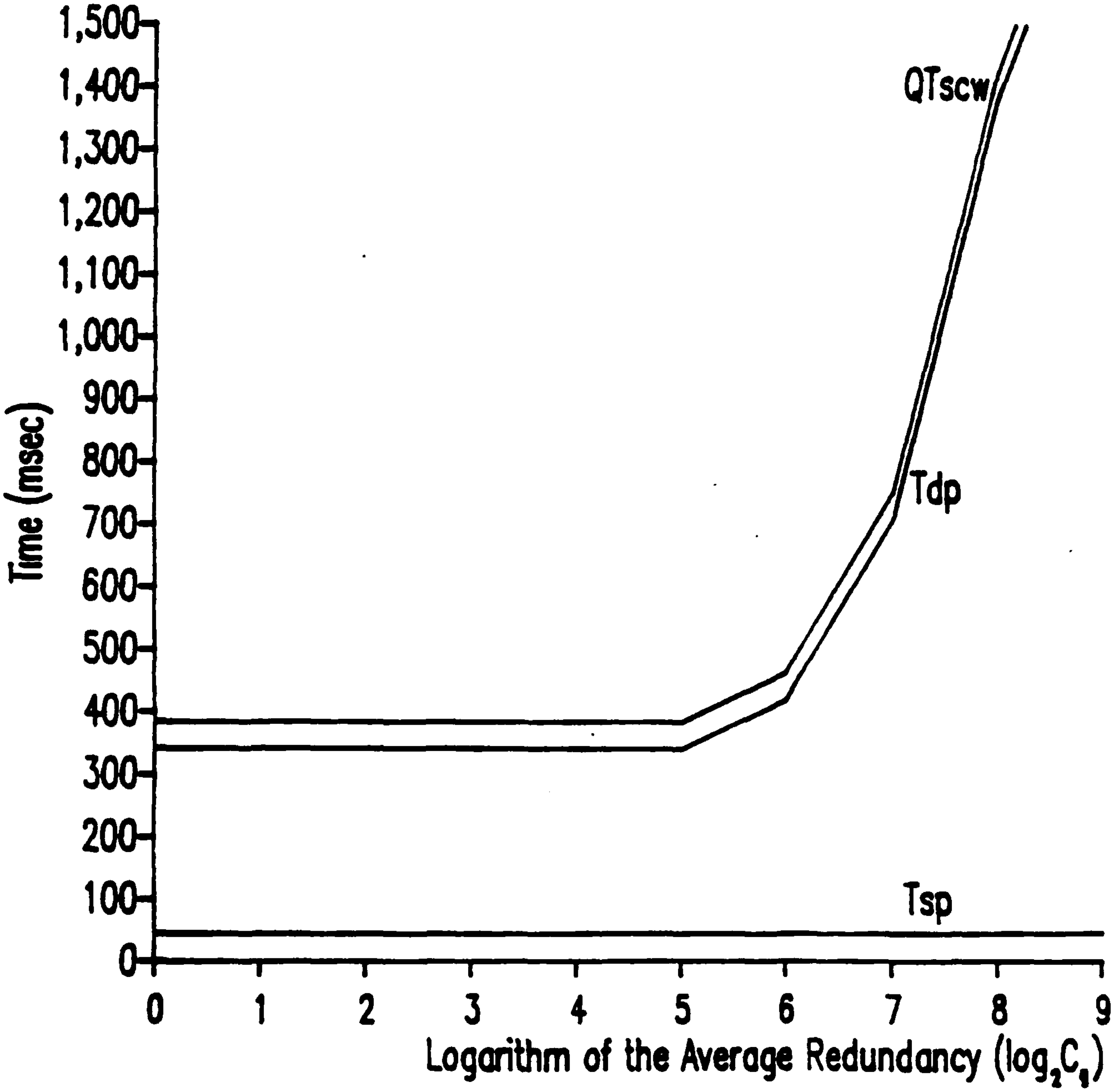


Figure 9.3.3 Components of the SCW Query Response Time (Sdb=10^5bytes, Ar=6, Rq=2, FD=9)

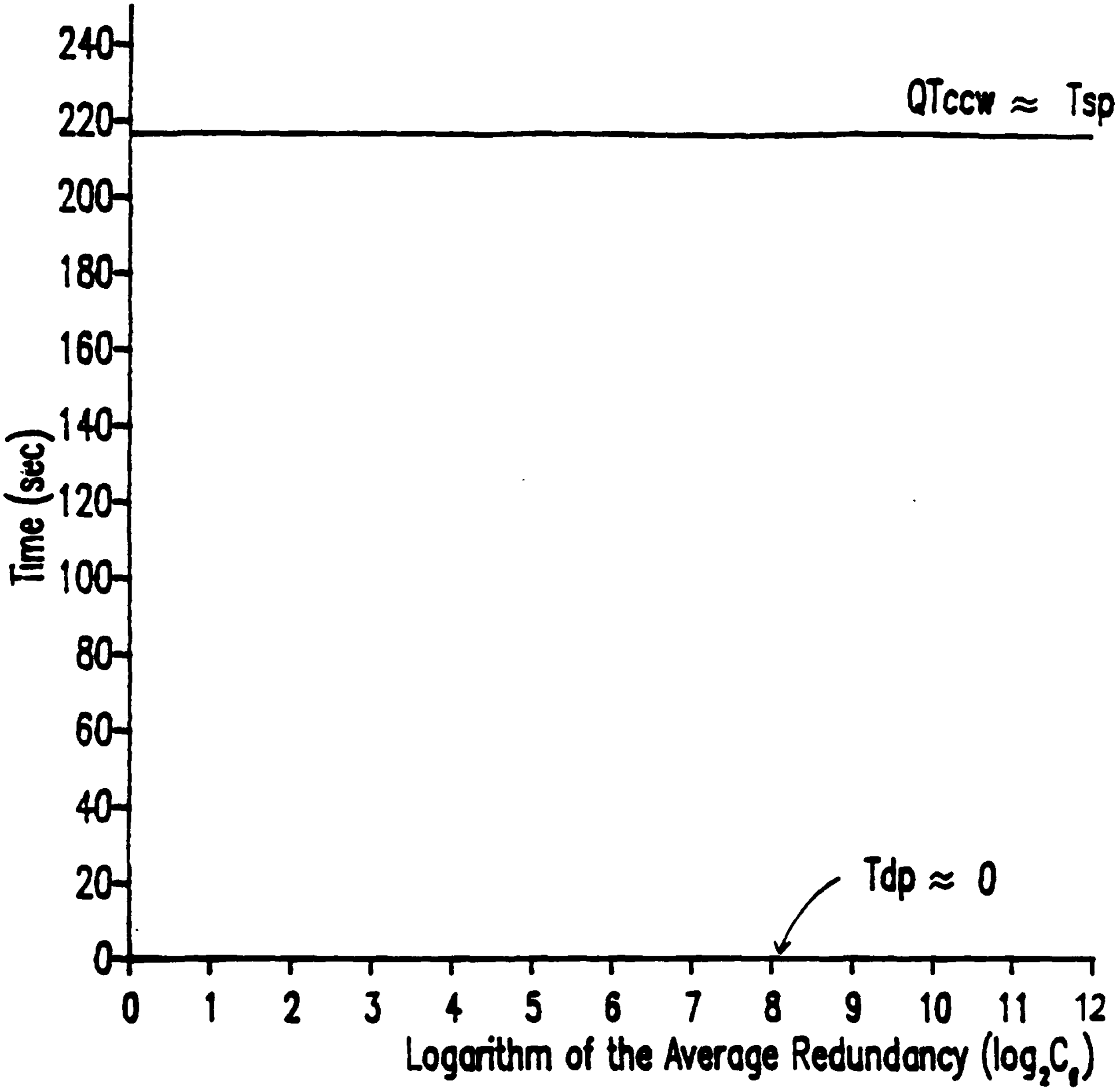


Figure 9.3.4 Components of the SCW Query Response Time
(Sdb=10^8bytes, Ar=6, Rq=2, FD=9)

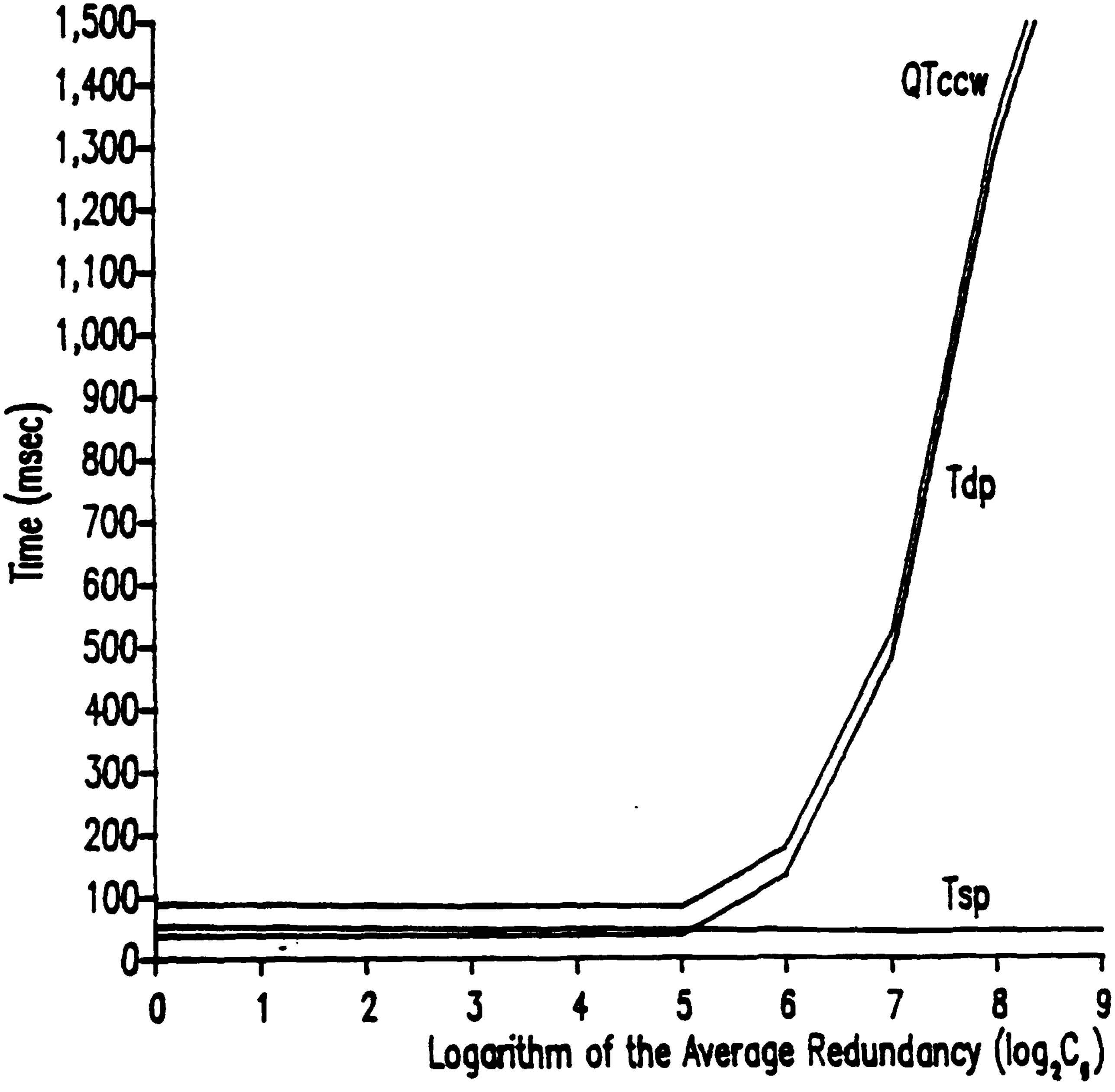


Figure 9.3.5 Components of the CCW Query Response Time
(Sdb=10^3bytes, Ar=6, Rq=2)

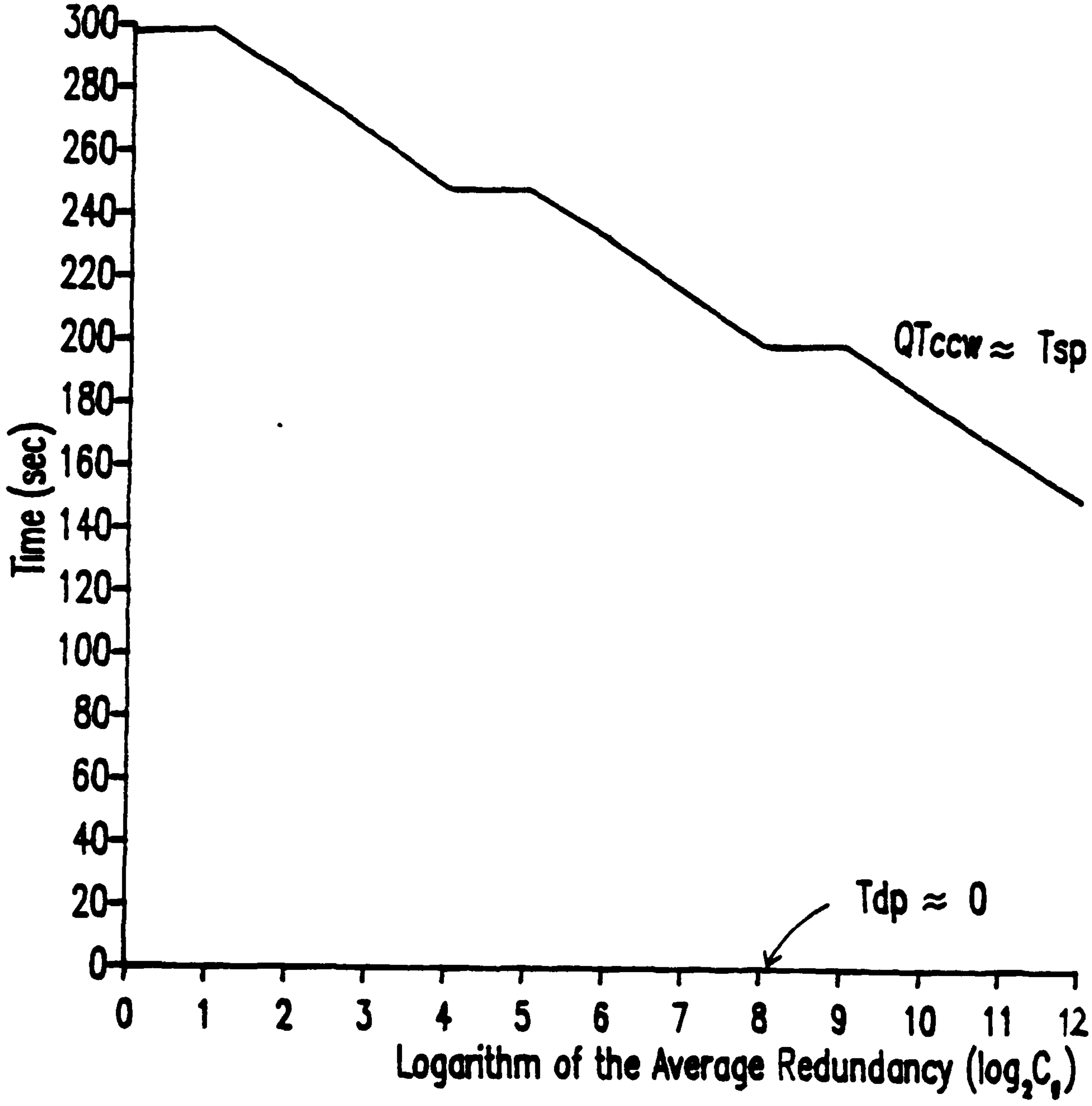


Figure 9.3.6 Components of the CCW Query Response Time (Sdb=10^9bytes, Ar=6, Rq=2)

9.3.3. Comparison of SCW and CCW Surrogate File Techniques

As shown by the simulation, the size and query response time of the CCW is smaller than those of the SCW when the average number of arguments specified in a query is small.

It is very easy to update SCW or CCW surrogate files. When a new tuple is added to the EDB, the corresponding code word is simply appended to the existing SCW or CCW surrogate files. No other operations are required. To delete a tuple, we must find and delete the entry in the surrogate file as well as in the EDB. When one changes the value of a field, SCW requires that a new code word be generated and the old one deleted. For CCW the change need only be made to the portion of the code word in question.

One obvious advantage of CCW over the SCW is that many relational operations can be easily performed on the CCW surrogate file rather than on the relations themselves [BER87]. This offers considerable potential savings in time to carry out those relational operations.

In SCW, the order of argument positions in either query or fact can't be differentiated because a SCW is generated by the logical OR operations on the BCW's. This property of SCW can be a disadvantage when used for rule indexing in the context of logic programming.

SCW surrogate file searching time can be reduced by using the bit-sliced organization to store the SCW files [LEE86]. But in that case, we must read and write back many blocks of SCW surrogate file to update one SCW, which is not tolerable when the EDB is dynamic.

In the SCW surrogate file technique, to reduce the the inherent false drops caused by the logical OR operations on the BCW's, one may assign different code weights to the BCW's of argument values depending on the occurrence frequency and query frequency of the argument values. But to do this, the code weights of frequently occurring argument values must be maintained in a table to be looked up whenever generating a binary code word [FAL85][ROB79].

9.3.4. Further Work with SCW and CCW

The main drawback of the SCW and CCW surrogate file technique is that the whole surrogate file must be read to the main memory and searched. To reduce the searching time, one can produce a block code word for each block of the surrogate file and use the block code words as an index for the surrogate file. A given QCW is compared with the block code words first and only those blocks of the surrogate file whose corresponding block code words match the QCW are retrieved and searched. But the speed up is achieved at the expense of the extra storage space and maintenance cost for the block code words. The performance of the block code words will depend on the following factors:

1) type of hashing functions used for code generation,

2) algorithm for generating the block code words,

3) blocking factor: number of code words blocked together to form a block code word,

4) how frequently the database will change.

[PFA80] introduced the block descriptor generated by logical Oring the disjoint codes of each tuple and [SAC83] considered two level superimposed coding scheme.

It has been shown that surrogate file processing time is dominant when the EDB is very large. Thus, if we adopt multiple processors and/or associative memory, we can reduce the surrogate file processing time considerably. A general structure of a back end system which contains multiple processors for the management of a very large extensional database of facts is shown in Figure 9.3.7. We assume that there are gigabytes of data stored on the EDB disks and there are gigabytes of CCW surrogate files stored on the SF disks. Suppose that the user is interested in retrieving fact data given some subset of values from a particular relation. The query code word would be constructed in the Request Processor using the proper hashing function and considering the positions of the values within the relation. The QCW would then be broadcast to all of the Surrogate File Processors (SFP's) to be used as a search argument. One could think of the SFP as a processor with associative memory with the QCW as the search argument. The SFP compares the QCW with each CCW and strips off the unique identifiers of matching CCW's. As soon as any unique identifiers are found by the SFP's they can be sent to the collector and passed on to the Extensional Data Base Manager (EDBM) for processing. The EDBM will retrieve the facts, compare them with the query to insure that a false drop has not occurred, put them in blocks, and send the blocks to the logic programming engine.

Furthermore, the SFP's can be extended to support complex relational algebra operations such as join. Consider a join using the hash join algorithm [BRA84],[KIT83]. Since the CCW surrogate files already consist of hash values, we only need to partition the portion of code words that represent the join variable and the associated unique identifiers into the SFP's. Then, the SFP's can perform the join operation independently. The associative memory in each SFP can be used for parallel execution of nested-loop join algorithm which outperforms the sort-merge join algorithm in a multiprocessor system [BIT83]. Based on matching within each SFP (which can be done in parallel), pairs of unique identifiers can be sent to the EDBM for final verification. Since the size of the CCW surrogate file is around 20% of the EDB, we can save a lot of time when we perform the relational operations on the CCW surrogate file rather than the EDB itself.

Since the CCW surrogate file technique can be implemented easily with multiple processors and associative memory to speed up the retrieval process and relational operations in a very large knowledge base system, our future research is towards the development of a special architectures supporting those CCW surrogate file techniques.

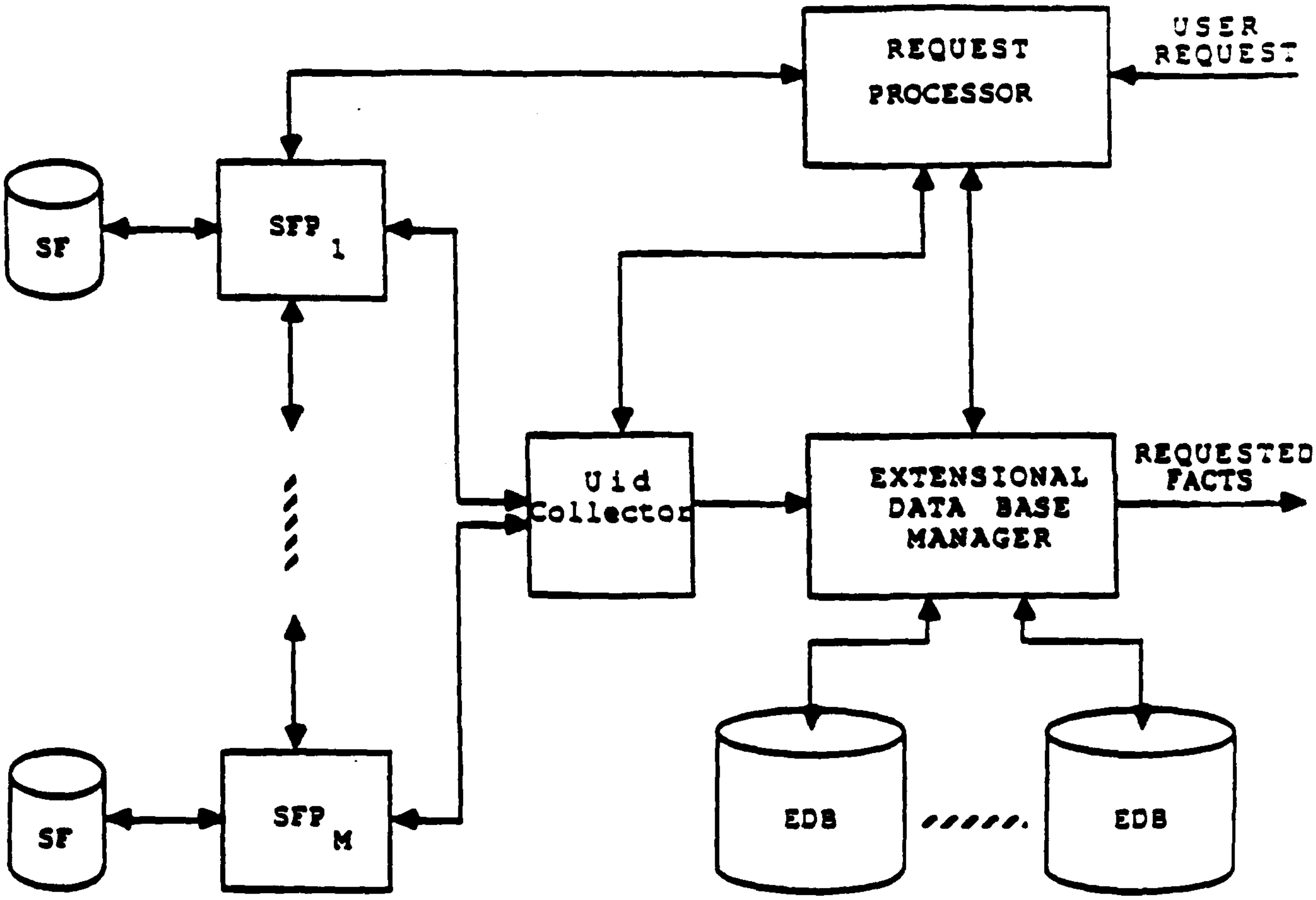


Figure 9.3.7 Back End System for Fact Management

9.4 Demonstration System for SCW and CCW

In this section the design of a demonstration system implementing the surrogate file concept for CCW and SCW is presented. The system is being developed on a VAX 8800, which serves as a frontend to a Connection Machine.

9.4.1 Demonstration System Design

The demonstration systems design is presented in Figure 9.4.1. The design can be viewed as a collection of subsystems tied together by the relational database management system INGRES.

9.4.1.1 INGRES

INGRES provides us with file management capabilities, which we would otherwise have had to write ourselves. There is the realization that by going through INGRES for our searches, there is a certain amount of incurred overhead, and thus the full advantage of the surrogate file cannot be realized on the demonstration system.

9.4.1.2 Logic Programming

A query enters the system from a logic programming environment. Once the query is received, it is passed to INGRES through an interface. In our system the interface is a Prolog one, being developed at Syracuse University. The interface transforms the Prolog query into a query (argument) that INGRES can manipulate. INGRES then passes the query to the Query Code Word Generator.

9.4.1.3 Query Code Word and Surrogate File Generators

In order to retrieve a fact, based upon one or more arguments, each argument is passed to the QCW Generator. The argument is hashed and a QCW is generated. What type of hashing is done depends on whether a CCW or SCW surrogate file will be used.

The Surrogate File generator forms the CCW and SCW surrogate files. As a new fact is entered through INGRES, it is passed to the surrogate file generator, where it is hashed according to whether a CCW or a SCW is being generated. Also, a unique identifier (UID) is generated. Both are passed to INGRES, which then passes them to the surrogate file.

9.4.1.4 Surrogate File and Knowledge Base

The respective CCW and SCW surrogate files are kept in the surrogate file area. During a retrieval operation the surrogate file is searched by INGRES, using the QCW as a primary key. For each match, INGRES will extract the UID. Once the UID is extracted from the surrogate file, it will be used to search the EDB.

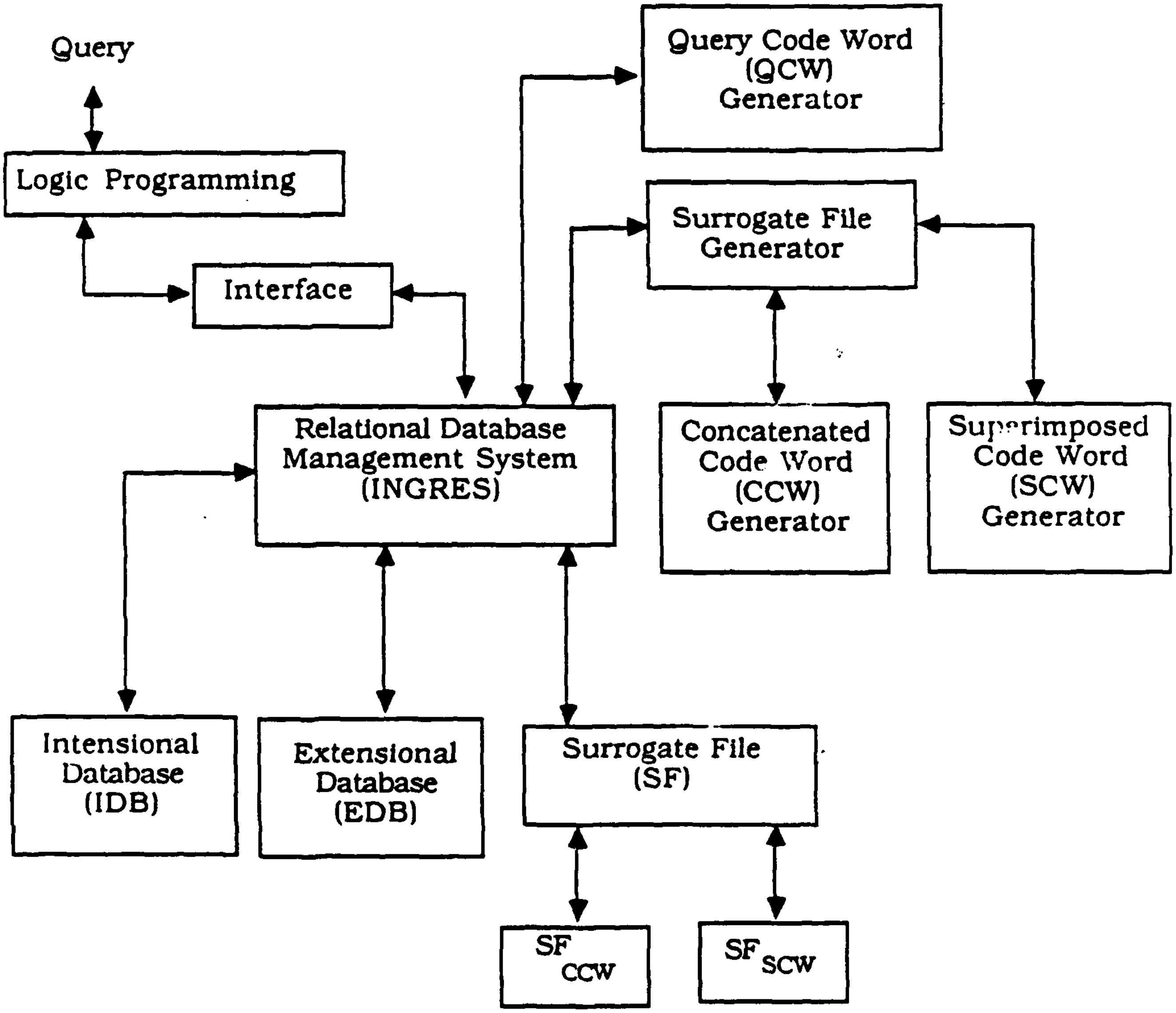

Query
Query Code Word (QCW) Generator
Logic Programming
Surrogate File Generator
Interface
Relational Database Management System (INGRES)
Concatenated Code Word (CCW) Generator
Superimposed Code Word (SCW) Generator
Intensional Database (IDB)
Extensional Database (EDB)
Surrogate File (SF)
SF CCW
SF SCW

Figure 9.4.1 Demonstration System for Surrogate File

9.4.2 Retrieving a Fact Using SCW

In Figure 9.4.2 we use SCW to visually explain how a fact will be retrieved from a SCW surrogate file in the demonstration system.

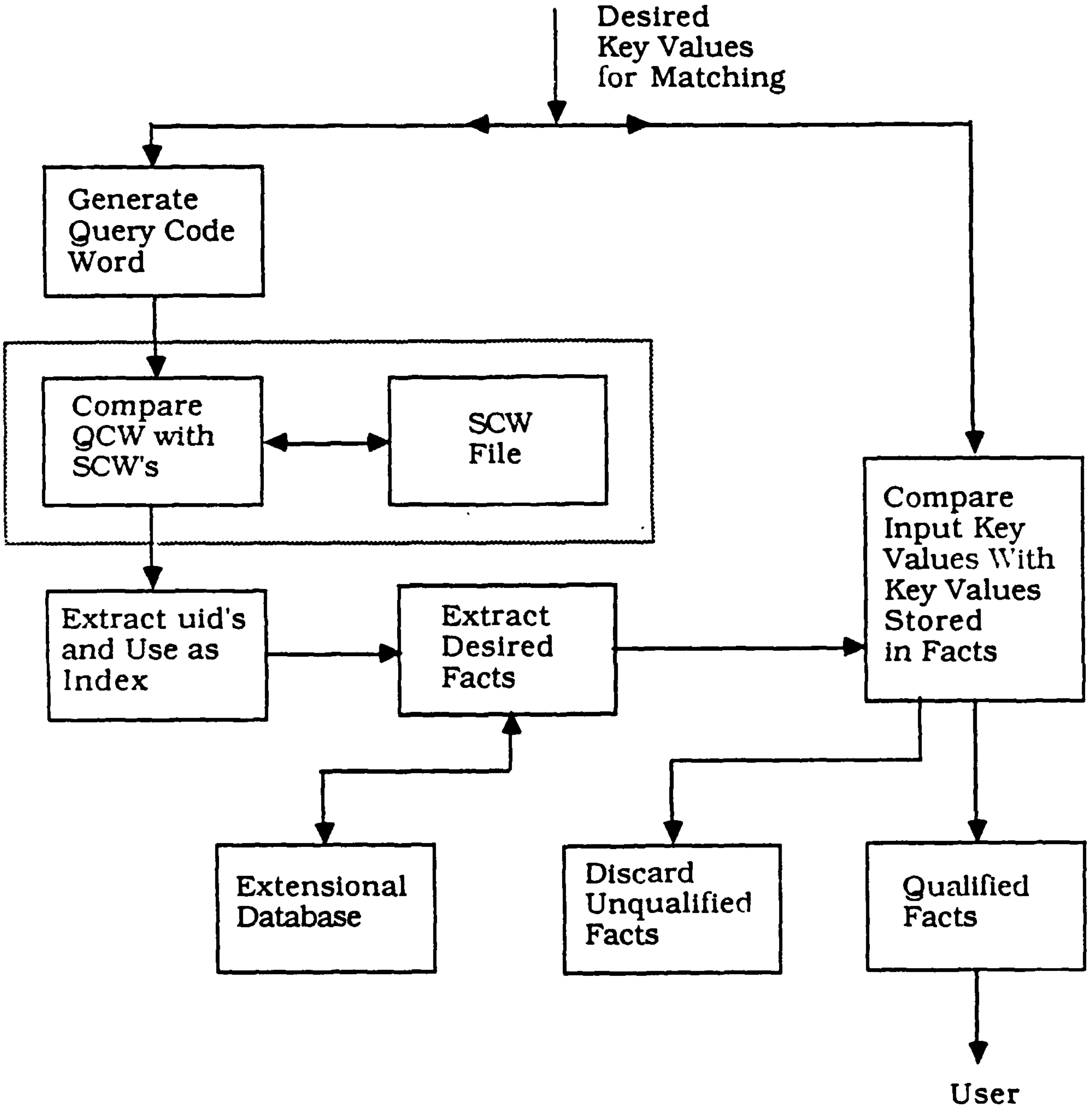


Figure 9.4.2 Partial Match Retrieval Using Superimposed Code Words (SCW)

9.4.3 Future Work on Demonstration System

The demonstration system will be developed using INGRES. The programming will be done in EQUEL. EQUEL supports both QUEL, the INGRES database programming language, and C in which the applications program will be written.

The first task will be to build an IDB and EDB. Having built them, the CCW and SCW surrogate files will be built. The next step will be to incorporate the Prolog interface into the demonstration system. As the work progresses, the plan is to in the future develop a parallel version of the surrogate file on the Connection Machine.

9.5. Inverted Surrogate Files

In this section we present transformed inverted lists, an inverted surrogate file structure for partial match retrieval applications. We show that TIL file structures are suitable for partial match queries on static files but with degraded performance and costly maintenance operations when dealing with volatile files. Then we extend the concept of inverted surrogate files to include dynamic files and orthogonal queries with the introduction of the Dynamic Random-Sequential Access Method (DRSAM) and Inverted Dynamic Surrogate Files (IDSF). Finally, we describe and analyze a parallel back end architecture for inverted surrogate files and discuss open research problems and future work.

9.5.1. System Model

Single or multilevel indexing is a common technique used in data base management systems (DBMS) for fast data access. In partial match retrieval, creating index files for more than one field in a record is necessary. The extreme case arises when every entry in a record is indexed independently and is referred to as inverted lists organization [DAT86]. The problem behind using inverted lists is that the size of the indices can become enormous, equal to or even larger than the data base size.

Transformed inverted lists (TIL) are similar to inverted lists with the main difference that indices are built based on the binary representation (BR) of the hashed output of a given field in a record of the data base relation. Two TIL types, TIL1 and TIL2, are considered in this section. A simple relation is illustrated in Figure 9.5.1. The fields are referred to as arguments and the BR values for argument position 2 are listed.

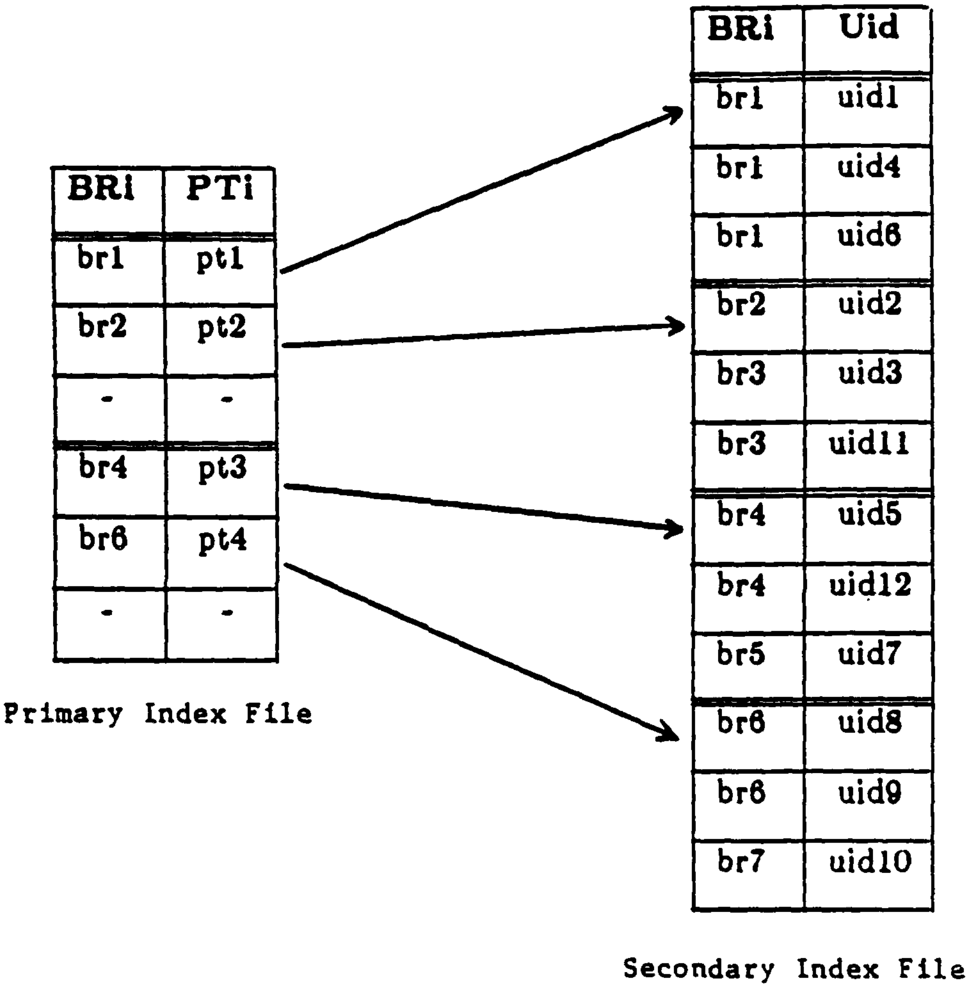
The application environment of the TIL technique would be the management of the EDB within a logic programming context. We assume that many different relations (fact types) with varying degrees and cardinalities exist in the very large extensional data base that we are considering. Furthermore, we assume that the tuples are stored in such a way that one first accesses the relation followed by an access to a particular tuple via its unique identifier (Uid). The unique identifier could be derived from the "primary key" of the relation or a serially generated number attached to each fact. Thus, the storage structure for the actual facts themselves would be very simple and a method such as extendible hashing [FAG79] or linear hashing [LAR82] could be used to guarantee retrieval of a given fact in at most two disk accesses. This presupposes that all secondary key retrievals will take place on the surrogate file or through post processing of the retrieved tuples if there are many different types of users of the same data base.

9.5.1.1. TIL1 Description

TIL1 consists of a two level indexed inverted list. Figure 9.5.2 illustrates the TIL1 organization for argument position 2 of the relation of Figure 9.5.1. The blank entries in the primary index file are usually included for updating purposes. The secondary index file for a given argument in a tuple is an ordered list of the BRs of the hashing function output of that argument with the attached unique identifier (Uid). The first entry in each block of this file is duplicated in the primary index file with an attached pointer to the corresponding secondary index block address. Furthermore, index files are partitioned in blocks of B bytes each.

Uid	Ar_1	Ar_2	Ar_3	Ar_4
uid1	......	br1	......	......
uid2		br2		
uid3		br3		
uid4		br1		
uid5		br4		
uid6		br1		
uid7		br5		
uid8		br6		
uid9		br6		
uid10		br7		
uid11		br3		
uid12		br4		

Figure 9.5.1 A Simple Knowledge Base Relation



BRi	PTi
br1	pt1
br2	pt2
-	-
br4	pt3
br6	pt4
-	-

Primary Index File

BRi	Uid
br1	uid1
br1	uid4
br1	uid6
br2	uid2
br3	uid3
br3	uid11
br4	uid5
br4	uid12
br5	uid7
br6	uid8
br6	uid9
br7	uid10

Secondary Index File

Figure 9.5.2 TIL1 for Ar_2 in Figure 9.5.1

It is observed that the entries in the primary index file are ordered as well.

When a given BR is to be retrieved (say BR=br3), the primary index file is sequentially accessed using the BR as the search argument and the pointer to the secondary block address corresponding to that BR retrieved (pt2 in our example). Then the secondary file is accessed in a direct mode and the required block(s) retrieved and searched sequentially for the occurrence(s) of the requested BR. The output is a list of Uids (uid3 and uid11 for our example) corresponding to the value of the request.

9.5.1.2. TIL2 Description

TIL2 is a three level indexed inverted list organization and is illustrated in Figure 9.5.3 for the same example relation. The difference between TIL2 and TIL1 lies in that the TIL1 secondary index file is now split into two files: the TIL2 secondary index file and the tertiary index file. Each entry in the tertiary index file consists of a Uid, so that the number of entries in this file is equal to the number of records in the data base relation. Each entry in the TIL2 secondary index file consists of three fields: the BR of the hashed function output of an argument value (say BR=br6), a list length entry "L" that provides the number of records in the data base that have the same entry value in a given argument position (2 for br6) and a pointer to the address of the first Uid in the tertiary file that has BR=br6. This pointer consists of the block address and a displacement value in the block.

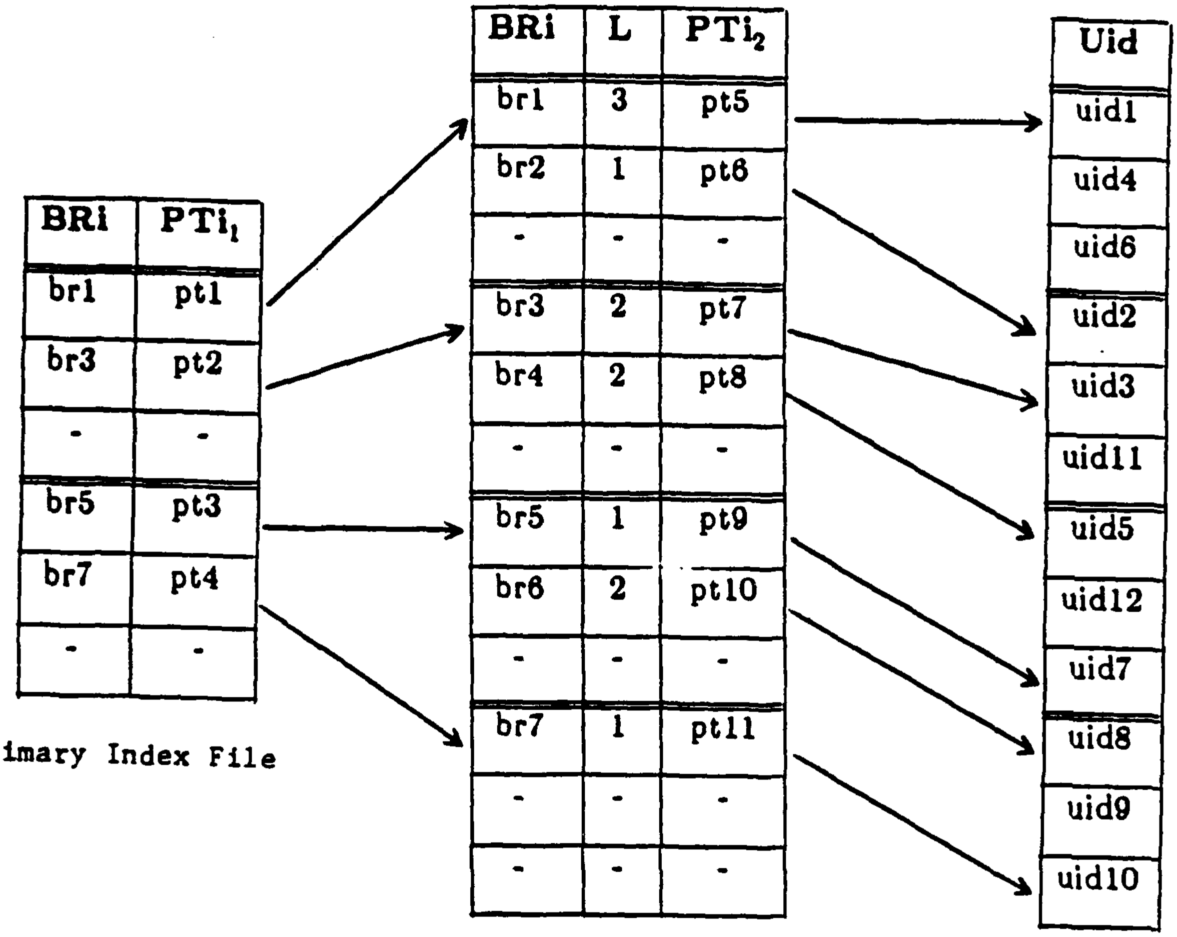
The retrieval process for TIL2 is similar to TIL1, but requires the access of an additional index level.

9.5.1.3. Partial Match on Multiple Argument Positions

When more than one argument position match is requested in a query, the different outputs from the inverted lists searches need to be intersected. The outcome of the intersection is a set of Uids that complies with the query requirements. Finally this set of Uids is used to directly access the main data base for the retrieval of the matched records. The gain in retrieval time when using transformed inverted lists is mainly due to the small size of the surrogate files and the fast access resulting from the indexing scheme. Only conjunctive partial match queries are considered, but the reader should be aware that disjunctive queries have the same level of complexity, with the lists intersection operation replaced by a multiple sets union operation.

It is noted that the inversion level of the surrogate files is determined by the application being considered. Since our underlying application involve logic programming and relational data bases, we assumed fully inverted surrogate files throughout and derived the minimum storage and the query response time equations in [HAC88]. Our analysis is based on a compact representation of the data and does not take into account overflow chains. It is meant to pinpoint performance bottlenecks, to be resolved in the design of a special purpose back end system.

The derived equations were based on the following general assumptions on the hardware and system models:



BRi	PTi_1
br1	pt1
br3	pt2
-	-
br5	pt3
br7	pt4
-	-

Primary Index File

BRi	L	PTi_2
br1	3	pt5
br2	1	pt6
-	-	-
br3	2	pt7
br4	2	pt8
-	-	-
br5	1	pt9
br6	2	pt10
-	-	-
br7	1	pt11
-	-	-
-	-	-

Secondary Index File

Uid
uid1
uid4
uid6
uid2
uid3
uid11
uid5
uid12
uid7
uid8
uid9
uid10

Tertiary Index File

Figure 9.5.3 TIL2 for Ar_2 in Figure 9.5.1

1. A given BR is equally likely to be specified in a query.

2. The primary and secondary indices are stored in contiguous secondary storage blocks and ordered with respect to the BR values so that a block can be searched in log time.

3. Buffer sizes are sufficient to hold the retrieved blocks and partial overlapping of the primary index blocks retrieval and search is achieved.

4. Main processor comparison is byte oriented.

5. We assume a stable file as defined in [LAR81] and do not account, in our deterministic analysis, for the overhead incurred by searching overflow records. According to Larson's stochastic model, the expected number of additional disk accesses required to search an indexed-sequential file is around 0.3 accesses.

6. The hashing functions do not lead to collisions. However, in practice, collisions could be deleted by post checking of the retrieved records from the EDB prior to further processing. This could be performed on the fly but is not included in the present analysis. Although not required for the analysis, if order preserving hashing functions are provided, [GAR86], TIL files could handle range queries as well.

We only present the results of our simulation pertaining to TIL1 and refer the reader to [HAC88] for additional details.

9.5.2. Simulation and Analysis of TIL Techniques

The notation definitions and parameters for the TIL technique are the same as those for SCW and CCW and are found in Table 9.3.1 and 9.3.2 respectively.

In Figure 9.5.4, the TIL1 surrogate file to data base size ratio is plotted versus the logarithm of the average redundancy factor, for different S_{db} and A_r values. In general the surrogate file size of TIL1 spans from a low of 9.2%, for $\log_2 C_g = 9$, $A_r = 10$ and $S_{db} = 10^5$, to 41.8% for $\log_2 C_g = 0$, $A_r = 2$ and $S_{db} = 10^9$. It is noted that the plots in Figure 9.5.4 mainly reflect the variation of the secondary index file size as the primary index file size can be shown to be negligible. In [HAC88], the storage requirements for TIL2 are reported to range from 8 to 20% of the size of the data base.

Figures 9.5.5 to 9.5.8 illustrate the TIL1 query response time (QT_{TIL1}) and its corresponding subprocessing times (T_{sp}, T_{it} and T_{dp}) for different data base sizes and number of arguments in a query. Figures 9.5.5 and 9.5.6 relate to medium sized files ($S_{db} = 10^7$ bytes) while Figures 9.5.7 and 9.5.8 are typical of very large files ($S_{db} = 10^9$ bytes). It is observed that QT_{TIL1} is highly dependent on the surrogate file processing time (T_{sp}) for low values of C_g (up to 512) and then becomes highly dependent on the intersection time (T_{it}). The drop in data base access time (T_{dp}), observed between the plots of Figures 9.5.5 and 9.5.7 or 9.5.6 and 9.5.8, is due to the dependency of the number of good drops (GD) on the ratio $\frac{C_g}{N}$. For a fixed C_g, this ratio decreases with increasing data base sizes.

No plots are included for the case where $R_q = 1$. In this situation, the query response time for TIL1 is dependent on the number of good drops which is C_g. Furthermore, TIL2 query response time variations are the same as for TIL1. The only difference is that TIL2 requires one additional disk access per query

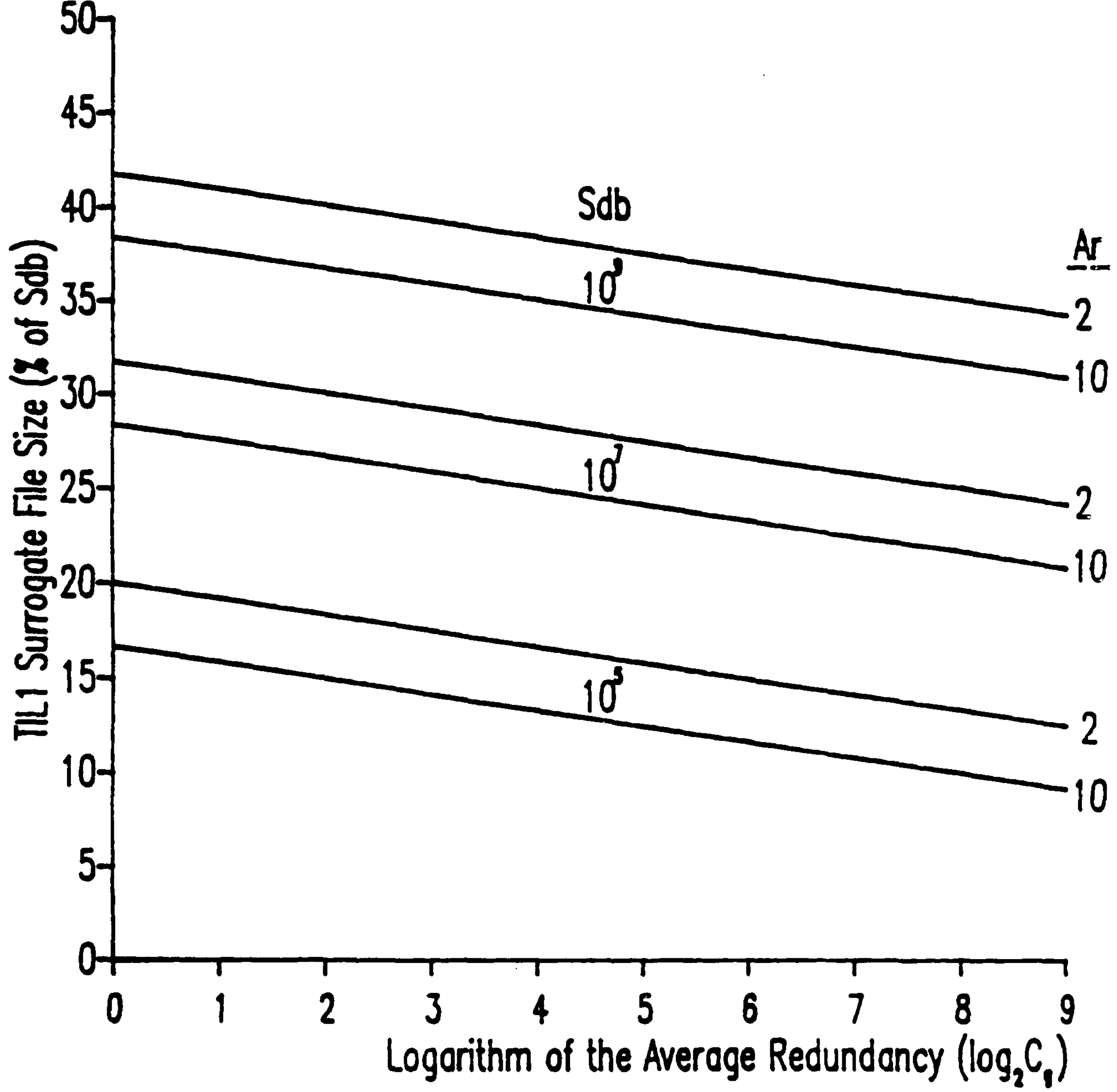


Figure 9.5.4 Effect of the Database Size and the Number of Arguments in a Tuple on the TIL1 Surrogate File Size.

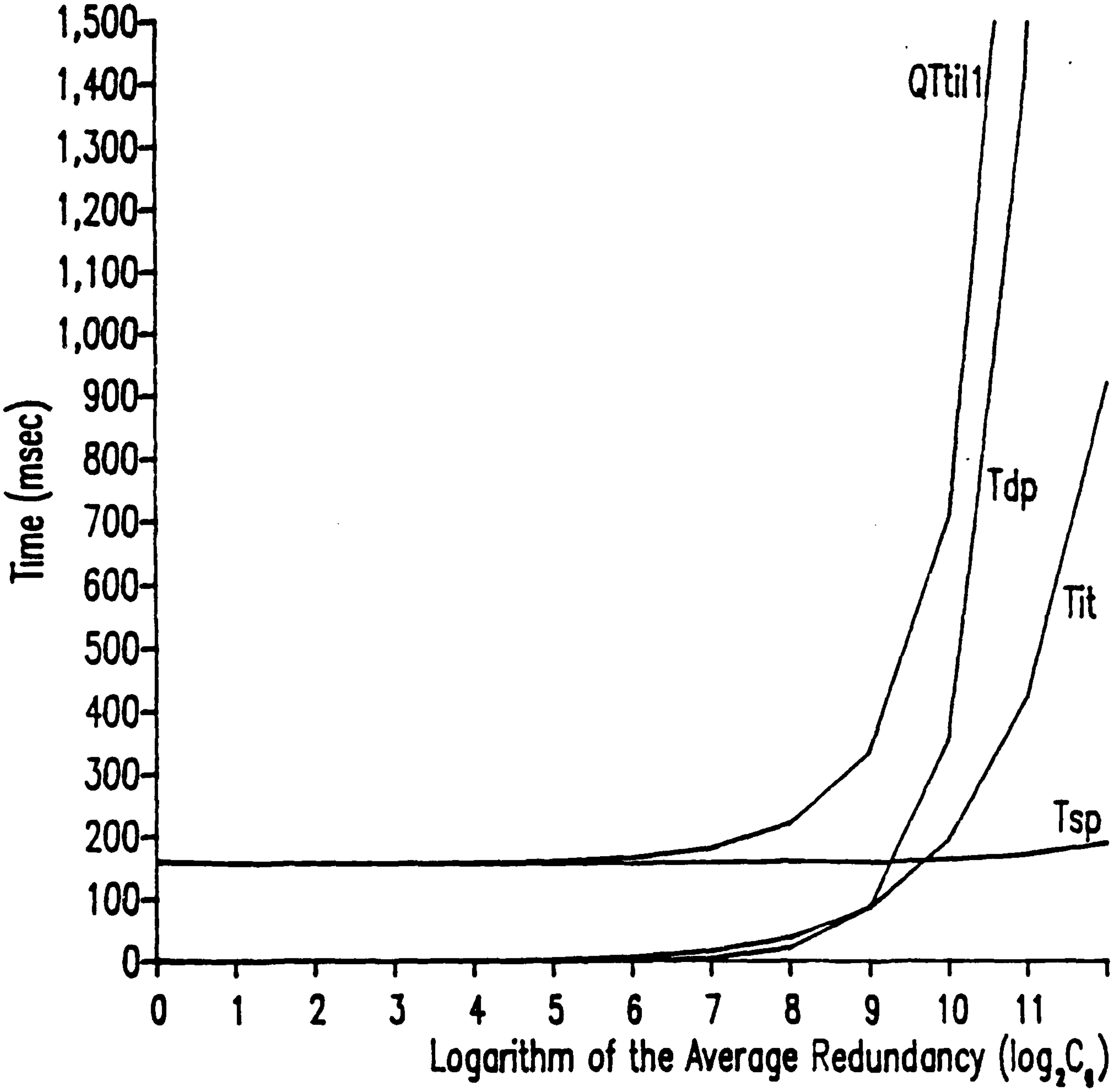


Figure 9.5.5 Components of the TIL1 Query Response Time
(Sdb=10^7 bytes, Ar=6, Rq=2).

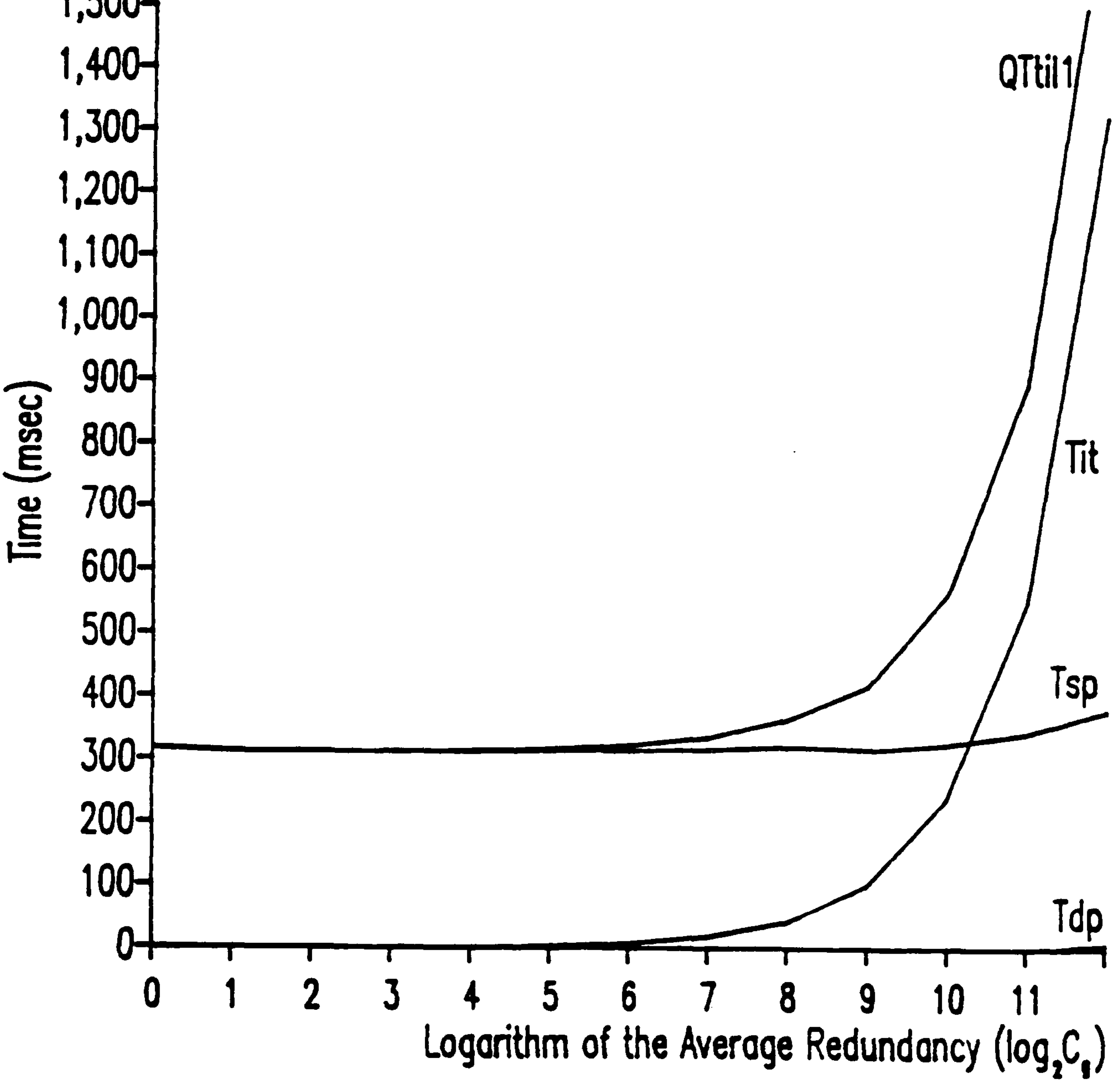


Figure 9.5.6 Components of the TIL1 Query Response Time ($Sdb=10^7$ bytes, Ar=6, Rq=4).

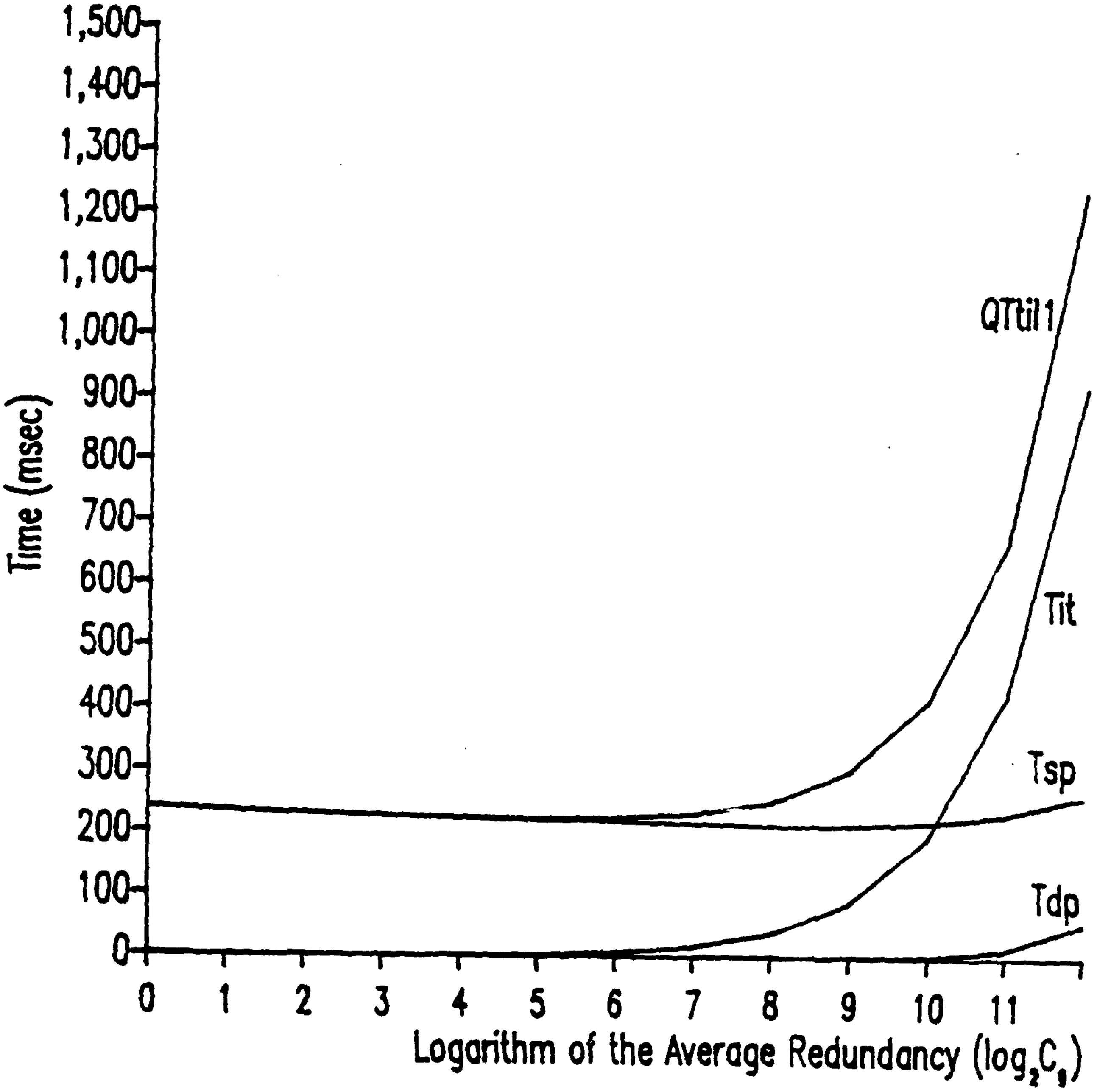


Figure 9.5.7 Components of the TIL1 Query Response Time (Sdb=10^8 bytes, Ar=6, Rq=2).

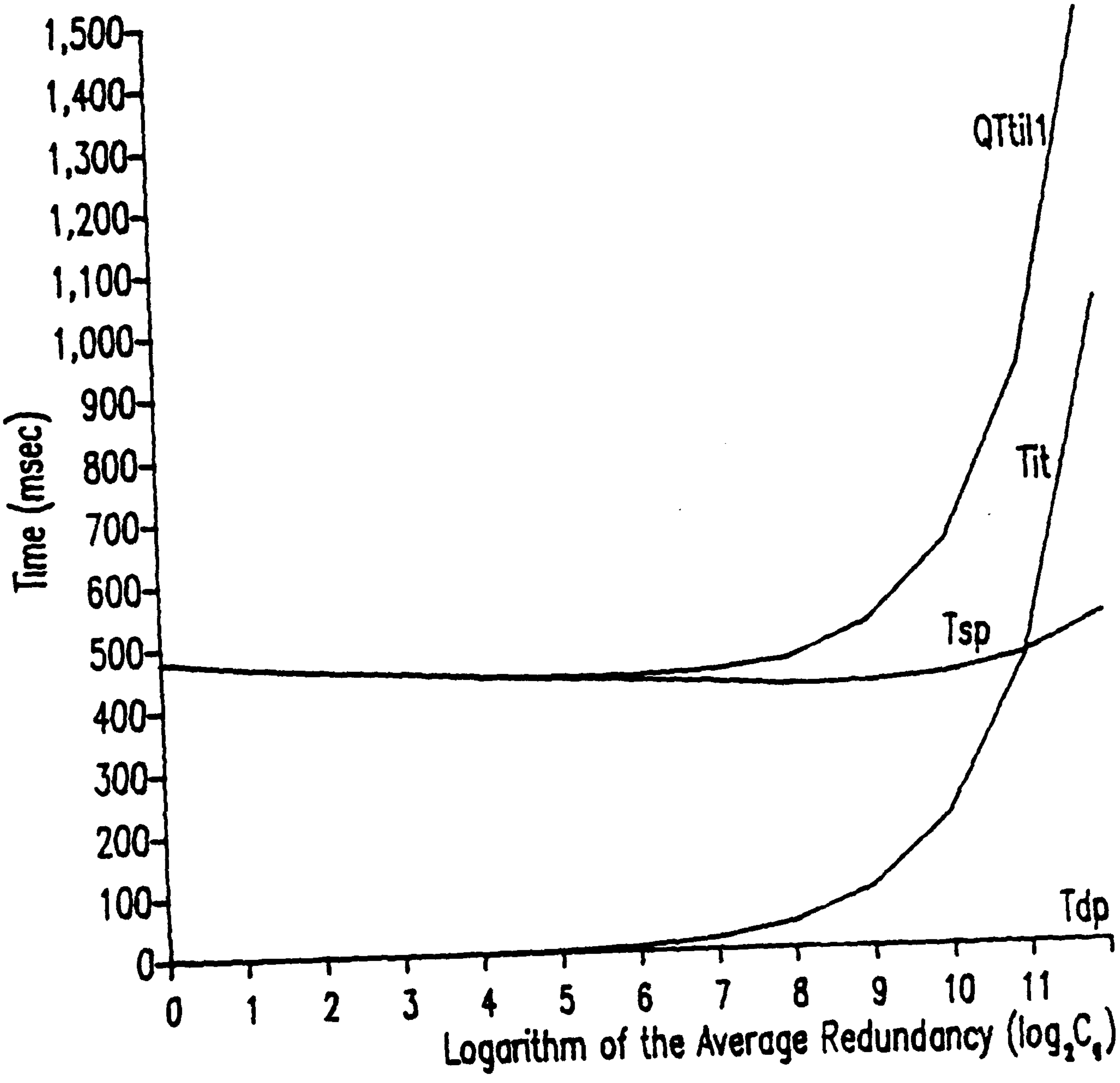


Figure 9.5.8 Components of the TIL1 Query Response Time ($S_{db}=10^9$ bytes, Ar=6, Rq=4).

argument, that is balanced by a smaller disk transfer time for large values of the redundancy factor C_g. The disk transfer time is smaller due to a smaller surrogate file size.

We conclude that the TIL techniques are efficient as to the storage/query response time combination. Even for relatively large redundancy factors, the query response time is within a few seconds while the storage overhead of the surrogate files lies in the 10 to 20 % range of the data base size. It is noted that conventional inverted lists, with full indexing, may require an overhead well in excess of 100 % of the data base size.

9.5.3. Maintenance Aspects of TIL Surrogate Files

One of the difficulties in using the TIL techniques is their maintenance requirements. Those become a serious drawback, especially in a highly volatile data base environment. The above analysis pertains to a static surrogate file. If, for example, 30% expansion of the main data base is forseen, the overall increase of the surrogate files sizes can be greater than 30%, due to the additional increase required for the different record pointers and unique identifiers.

Some important maintenance aspects are the add, delete and update operations. When adding a new record to the data base, all the index files have to be accessed and reordered; which is a time consuming operation. The use of overflow blocks would decrease the time requirements for the insert operation with a negative impact on query response time. Block inserts could be followed but this technique is not applicable to real time data bases. In any case, periodical time consuming reordering is necessary. Deleting records could be performed by marking techniques and delaying reordering and packing operations to off line maintenance periods. Finally, updates require the access and rearrangement of the affected attribute's indices.

It can be stated, in general, that the overall management system requirements for TIL surrogate files is complex and those techniques are not recommended in volatile data base environments.

Provided order preserving hashing functions, orthogonal queries are possible with inverted surrogate files. With the additional requirement to manage very large dynamic data/knowledge bases, we are led to the topic of our current research which we present in the next Section.

9.5.4. The Dynamic Random-Sequential Access Method

The scope of our work is to extend the concept of inverted surrogate files to cover the more interesting and general case of dynamic data/knowledge bases. A new dynamic file structure is proposed, as the core structure for inverted surrogate files. Furthermore, we propose the analysis and simulation of this structure and the development of a back end architecture based on inverted dynamic surrogate files for the management of a Very Large Data/Knowledge Base (VLDKB).

Two major dynamic hashing schemes are exhaustively analyzed in the literature, namely extendible hashing (EH) by Fagin [FAG79] and linear hashing (LH) by Litwin [LIT80]. While the basic schemes of EH and LH were proposed for primary key direct access applications, those were modified, extended and adapted for a wider range of file design problems, including PMR. Most applications related to PMR follow the multi-attribute single file design approach. The Dynamic Random-Sequential Access Method (DRSAM) is proposed to be used as the core structure for a single attribute multi-file design. It is inspired from LH with the additional feature that the ordered sequential characteristic of inverted files and TIL is preserved for optimum sequential processing (range queries) as well as random access.

This scheme has the same near optimal characteristics of LH as to random access, insertion, deletion and update operations and the additional important feature of fast sequential access similar to ISAM, VSAM [MAR77] and B-trees [BAY71] with an O(1) response time for sequential access. The analysis is kept to a "primary key" file, and within some constraints to be discussed, the reader can easily check that DRSAM is applicable to secondary keys as well. Furthermore, the analysis assumes a contiguous storage allocation scheme. The case of a distributed secondary storage allocation environment shall be covered in future work.

9.5.4.1. A Review of Linear Hashing

The reader is referred to the paper by Litwin [LIT80] for additional details on LH. Linear hashing is a directoryless dynamic hashing method and relies on a one sided linear expansion of the file following a sequential bucket split pattern.

The basic idea is best explained by an example: assume that we start with a file of 4 buckets (#0 to #3), each with the capacity to store 3 records and the hashing function that determines the address of a key given by h_0(Key) = Key mod 4 (h_0 is called the home hash function). Initially the file is loaded with 10 records as shown in Figure 9.5.9. We note that buckets #1 and #2 are full. We assume that the file expands whenever a collision occurs (referred to as uncontrolled splitting): a collision takes place when a new record's key, to be inserted, hashes to a bucket that is already full.

The expansion of the file is performed by extending it through the addition of one bucket at a time. This bucket receives some records moved from an existing bucket that undergoes a split (i.e expands). The next bucket to split is determined by a pointer (called split pointer) that moves sequentially, after each split, from bucket #0 to bucket #3. The file gradually grows from 4 to 8 buckets (#0 to #7) and the process of doubling the size of the file is referred to as an expansion cycle. At the beginning of an expansion cycle, the split pointer points to bucket #0 (marked by "*" in Figure 9.5.9). The split is resolved by rehashing the splitting bucket's records with h_1(Key) = Key mod 8 (h_1 is referred to as the split hash function).

#0 (*)	#1	#2	#3				
0	101	10	3				
60	201	70	7				
	205	130					

Figure 9.5.9.

Let us insert Key=134: h_0(134)=2 and a collision occurs. Key 134 is inserted in an overflow area for bucket #2 and bucket #0 splits: the records in bucket #0 are rehashed with h_1, moving Key=60 to the new bucket #4. Then the split pointer is advanced to point to bucket #1 and we get the file status of Figure 9.5.10.

#0	#1 (*)	#2	#3	#4			
0	101 201 205	10 70 130	3 7	60			

Overflow Area

		134					

Figure 9.5.10.

When the file doubles in size the new home hash function is set to h_1 = Key mod 8. A new expansion cycle can begin with the split hash function as h_2 = Key mod 16. The split pointer is reset to bucket #0 and the new cycle will expand the file from 8 to 16 buckets.

In gene. al, to implement linear hashing, starting from a file of "N" buckets, we need a sequence of hashing functions (h_0, h_1..., h_i, h_{i+1},...) with the following properties:

$$0 \leq h_0(\text{Key}) \leq N-1$$

$$h_{i+1}(\text{Key}) = \begin{cases} h_i(\text{Key}) \\ \text{or} \\ h_i(\text{Key}) + N \times 2^i \end{cases} \qquad \text{for all Key and } i \geq 0$$

The simple remainder hashing function is one which has the above property. To achieve an even load, the two cases for h_{i+1} should occur with equal probabilities. To keep track of the state of the file, two variables are needed: "L" counts the number of times the file size has doubled and "p" as the split pointer to the next page to split. The address computation algorithm is as follows:

```
address(Key) = $h_L$ (Key);
if (address(Key) < p) then address(Key) - $h_{L+1}$ (Key);
```

Expanding the file by one page requires the local reorganization of two pages: the one being split and the new page appended to the end of the file. The technique outlined above gives a mechanism to expand the file by one page. The criterion to trigger an expansion was based upon the occurence of a collision. This mechanism is referred to as "uncontrolled splitting". Litwin suggests "the rule of constant storage", whereby the designer would set a threshold for the storage utilization: whenever this threshold is exceeded the file is expanded by one page. This method is also referred to as "controled splitting".

As an indication of the performance that can be achieved with LH, Larson [LAR82] reports the following: a page size of 20 records, storing overflow records on overflow pages with a capacity of 6 records per overflow page, and a threshold of 0.85 on the storage utilization result in an average successful record retrieval in 1.26 disk accesses, and an average record insertion cost of 3.49 accesses (including disk accesses for file expansion).

It is noted that the split sequence follows a sequential pattern from the first to the N-th bucket. This means that the split does not necessarily take place on the bucket that undergoes a collision, which is typical of directoriless dynamic hashing methods. A collision resolution method (CRM) is proposed to resolve the split by assigning overflow chains. If the data is uniformly distributed the performance of the file structure is not degraded by the overflow chains.

As shall be seen, DRSAM relies on a different split sequence that achieves clustering for fast sequential access as well as an expected random access performance equal to LH.

9.5.4.2. File Design Objectives

The objective is the design of a file structure with the following characteristics:

1. Fast random access: the structure should be such that, given a search key, the access cost to the required record is optimal, i.e one disk access (or very near to the optimal value of one).
2. Fast sequential access: the structure should be such that, given a range for a search key, the access cost required is also one disk access followed by successive block reads, provided a contiguous storage allocation scheme, or optimal for distributed allocation schemes.
3. Dynamic: the structure should be easily expandable with low maintenance overhead.

Characteristics 1 and 3 are studied in the context of LH. Characteristic 2 is achieved if the buckets that qualify for the range query are located in contiguous blocks in a sequential allocation environment, so that one disk access is performed followed by consecutive bucket reads, or the number of disk accesses is minimized in a distributed allocation environment. This is typical of the fairly static ISAM and VSAM files in general. Also, B-trees provide fairly linear results for sequential access and log time for random access. DRSAM is based on an order preserving hashing function with a one sided expansion scheme similar in concept to LH but follows a different split pattern, designed to preserve the natural order of the key values in consecutive blocks. Its random access cost is the same near optimal one as LH and its sequential access cost is expected to be O(1). This method presents a promising alternative to the static ISAM, VSAM and B-trees file structures.

9.5.4.3. Order Preserving Hashing

The hashing scheme is as follows: the file is at level "i" would mean that it consists of 2^i (contiguous) buckets. The address of a "Key" would be found by transforming "Key" with $OPH_i(Key)$, where OPH_i is a dynamic sequential allocating and order preserving hashing function:

$$add_i(Key) = OPH_i(Key)$$

at level "i+1", we write:

$$add_{i+1}(Key) = OPH_{i+1}(Key)$$

A simple dynamic order preserving function is provided for $OPH_i()$ as :

$$OPH_i(Key) = Prefix(Key,i)$$

with Prefix(Key,i) as the leftmost "i" bits of "Key". We chose this function as being easy to follow, though not generally considered as a good randomizing function. Other randomizing functions could be devised and the reader is referred to Garg's work [GAR86] for order preserving hashing and Carter et al. [CAR79].

Assuming the contiguous allocation scheme, the reader can easily check that we indeed have an order preserving hashing function. This function is simple and provides a fast mean to compute the address of a key. Furthermore, if a key range is provided, the blocks to be retrieved lie in "contiguous" blocks whose addresses are linearly found by hashing the extremes of the range (assuming that all blocks covering the range of the query are on the same level).

It can be easily checked that the above sequence of hashing functions does not lie in the class of linear hashing split functions advocated by Litwin. Therefore we propose a different split pattern (which we refer to as the "one sided logarithmic folding"). The split algorithm is described in Section 9.5.5 while the following section contains an example to provide an insight to the expansion pattern of this file structure. It is observed that, like LH, the expansion should be on one dimension as operating systems cannot easily cope with files that expand in two directions.

9.5.4.4. An Example of the Expansion Pattern of DRSAM

In the following pictorial representation, we assume block sizes of b=3 records and the home hash function is $add_2(Key) = OPH_2(Key)$, i.e N = 4 buckets or the current level "i" is 2. Then the split hash function is $add_3(Key) = OPH_3(Key)$ and we are looking at the three leftmost bits. Assuming that a key is encoded in 8 bits, the ranges for level 2 are as follows:

bucket #0: 0 to 63.

bucket #1: 64 to 127.

bucket #2: 128 to 191.

bucket #3: 192 to 255.

For an expansion cycle from level 2 to level 3, each range splits in consecutive buckets. For example, bucket #0 splits onto buckets #0 and #1 and the respective range is then: 0-31 and 32-63, and so on. In general, bucket #x splits onto buckets #2x and #(2x+1).

In Figure 9.5.11, the state of the file is shown with 9 insertions. We observe that bucket #0 is full. We begin with the split pointer at N/2= bucket #2 (marked by a "*").

#0	#1	#2 (*)	#3				
0	70	130	200				
10	72	162	235				
60							

Figure 9.5.11.

Figure 9.5.12 shows the file state after the insertion of Key=12. This value hashes to bucket #0 and a split occurs with an overflow chain attached to bucket #0. Bucket #2 splits onto buckets #4 and #5. Note that bucket #2 is not used for the moment. We shall refer to it as the "hole". This hole expands and shrinks during the expansion cycle. During an expansion cycle, the maximum number of buckets that would be unused at a given time can be shown to be $\log_2 N = i$. This is one of the drawbacks of this technique and represents the price we have to pay to preserve the desired sequential access characteristic of the file. We shall further talk about the "hole" characteristics in the following section. The split pointer is advanced to bucket #3.

#0	#1	#2	#3 (*)	#4	#5		
0	70	-	200	130	162		
10	72	-	235				
12		-					

Overflow Area

60							

Figure 9.5.12.

Let us see what happens with the successive insertions of 120, 131, 121, 122 and then 62: first 120 goes in bucket #1, then 131 in bucket #5 (as bucket 2 has already split and is at level 3 now). Figure 9.5.13. shows the status of the file at this stage.

#0	#1	#2	#3 (*)	#4	#5		
0	70	-	200	130	162		
10	72	-	235	131			
12	120	-					

Overflow Area

60							

Figure 9.5.13.

Then comes 121, a collision occurs and bucket #3 splits onto buckets #6 and #7. The bucket split pointer "folds back" to bucket #1 as the consecutive buckets #2 and #3 are now empty and can be used to expand bucket #1. Figure 9.5.14 shows the state of the file after inserting Key=121.

#0	#1(*)	#2	#3	#4	#5	#6	#7
0	70	-	-	130	162	200	235
10	72	-	-	131			
12	120	-	-				

Overflow Area

60	121						

Figure 9.5.14.

With 122 inserted, bucket #1 splits on buckets #2 and #3 and the split pointer folds back to bucket #0 as shown in Figure 9.5.15.

#0 (*)	#1	#2	#3	#4	#5	#6	#7
0	-	70	120	130	162	200	235
10	-	72	121	131			
12	-		122				

Overflow Area

60							

Figure 9.5.15.

Finally, inserting Key=62 induces a collision and bucket #0 splits onto buckets #0 and #1. At the end of the process, the file has undergone a full expansion cycle and is at level 3. The split pointer is advanced to bucket #4 and a new expansion cycle can begin. The status of the file is shown in Figure 9.5.16.

#0	#1	#2	#3	#4 (*)	#5	#6	#7
0	60	70	120	130	162	200	235
10	62	72	121	131			
12			122				

Figure 9.5.16.

The reader can easily determine that the resulting load factor is low (0.625). The load factor is expected to be similar to LH, and with an uncontrolled split mechanism Litwin reports a load factor that is lower than the one of EH (around 0.60). Controlled splitting techniques could be applied as well as Larson's partial expansion method [LAR80] to improve on the load factor keeping the near optimal direct access performance. It is observed that Larson's partial expansion method is not applicable as is, but should be modified to capture the sequential order of the file.

The splitting sequence is not easily seen from the example but the algorithm in the next section provides an elegant and simple solution to the computation of the address of the next bucket to split.

9.5.5. Underlying algorithms for DRSAM

In this section we describe the underlying algorithms that control DRSAM. Those algorithms are described following a pseudo_C notation and are based on the uncontrolled splitting mechanism.

1. Address Computation Algorithm

For an insertion or a search operation, the bucket address of a record is determined in a similar way as for linear hashing and is given as follows:

```
buck_add(Key)
{
l = i; /* set level to be the home level "i" */
       * "m" is the home bucket address of Key */
m = $add_l$(Key);
/* Check the level of the computed address using the procedure
 * Level(m) described in Section 4.3. Level(m) determines if we
 * need to rehash with $OPH_{i+1}$ to compute the address of Key */
l = Level(m);
if (l == i+1)
       {
       m = $add_l$(Key);
       }
return(m);
}
```

The bucket address computation is quite simple, provided that the level of the record can be determined with a fast routine. For LH, the level is determined with one comparison step while, as shall be seen, our method is slightly more complicated but still computable in a straightforward manner. This is a required computation overhead to keep the sequential access characteristics of the file.

2. Algorithm for the Next Bucket to Split

As previously stated, a bucket "x" always splits onto buckets "2x" and "2x+1". The split pattern is as follows: we begin by splitting bucket N/2 onto N and N+1, then N/2 +1 onto N+2 and N+3, followed by a fold back to N/4 onto N/2 and N/2 +1, then back to N/2 +2 onto N+4 and N+5 ... The strategy is to know when to "fold back" and use the emptied space efficiently: as a general rule, a "fold back" takes place when two consecutive buckets are emptied through previous splits. While this pattern may seem complicated, the algorithm we provide is very simple and straightforward.

"N" is the number of buckets in the file for level "i": $N = 2^i$. "Pt" is the pointer to the next bucket to split in the upper range N/2 to N-1 and Splitpt is the pointer address of the bucket that will undergo a split.

Initially: Pt=N/2 + 1; Count=0 and Splitpt=N/2;

```
void split()
{
/* step 1 */
"Perform split by reading bucket pointed to by Splitpt and rewriting the
2 resulting groups on the consecutive buckets given by OPH$_{i+1}$(Key)";

/* step 2: this step tests for a "fold back condition" and assigns
 * the next split bucket address */
if (Splitpt is odd)
{
 Splitpt = (Splitpt -1)/2;
}
else
{
 Splitpt=Pt; Pt=Pt+1;
}
/* step 3: test for a completed expansion cycle */
if (Splitpt == N) i = i +1;
return;
}
```

It is noted that the expansion is natural, and at the end of an expansion cycle, "Splitpt" points to "N". We only need to set the home level to "i+1". This is done with step 3.

The split sequence for the expansion of a file of 16 buckets to a file of 32 buckets (level 4 to 5) is shown in Figure 9.5.17.

Bucket #	0	1	2	3	4	5	6	7
Split #	16	15	7	14	3	6	10	13

Bucket #	8	9	10	11	12	13	14	15
Split #	1	2	4	5	8	9	11	12

Figure 9.5.17.

As compared to LH, the split sequence for a contiguous file is two splits followed by one fold back split, then two splits followed with two fold back splits and so on. It is noted that fold back splits are made to reuse the emptied buckets by previous splits. Empty buckets appear in the file structure at the beginning of an expansion and disappear at the end of the expansion cycle. It is easy to show that, during an expansion cycle, this "hole" can consist of atmost "i" unused buckets". The reason for the "hole" is that the splits use two "new" physical buckets instead of one bucket as for LH and subsequent splits tend to increase its width (in scattered but traceable locations). One would think that this is going to affect the load factor of the file. For large files, the "hole" would not be of importance as to its effect on the load factor.

For a full expansion from 16 to 32 buckets, the sequence for the number of empty buckets that compose the hole is: 1,2,1,2,3,2,1,2,3,2,3,4,3,2,1,0 buckets. This would take place if we assume that we do not use the emptied slots until a collision occurs or, for controlled splitting, until a certain load factor is achieved. The effect of the hole needs further investigation.

While for small files the hole leads to a poor load factor, its effect becomes negligible as the file grows in size. As a first qualitative evaluation, the maximum offset between the load factor of LH and DRSAM is equal to $\frac{1}{2^i}$. this relative value becomes negligible if we are dealing with very large files and would not considerably affect the storage overhead or the response performance of DRSAM.

We still have to devise how to recognize which bucket is at level "i" and which is at level "i+1", during an expansion cycle. This is necessary to compute the exact bucket address for an insertion, deletion or search operation.

3. Determining the Level of a Key

Determining the level of a key is, to a certain extent, the inverse of the bucket split address algorithm. We have to determine, within the range of the key, if the pointed to bucket has undergone a split. Fortunately, this is done in an elegant way as well with the procedure Level(m), where "m" is $OPH_i(Key)$ and is the home bucket address of "Key" for level "i". Let us first put down some mathematical formulae needed to clarify this procedure:

Define $\beta = \frac{1}{2^{(i-1-\lfloor \log_2 m \rfloor)}}$, with $m \neq 0$. The special case $m = 0$ is accounted for on its own in the algorithm. Let $\gamma_l = \beta \times \frac{N}{2}$ and $\gamma_h = \beta \times N$, then we can write: $m \in [\gamma_l, \gamma_h[$ with the interval being the "range of m". The number of splits that occurred in the range of "m" is denoted NS_m and can be easily shown to be:

$$NS_m = \begin{cases} \lceil (Pt - N/2) \times \beta \rceil - 1 & \text{if } Splitpt \in [\gamma_l, Pt[\\ \lceil (Pt - N/2) \times \beta \rceil & \text{otherwise} \end{cases}$$

Let $m' = m - \gamma_l + 1$, then the Level(m) procedure is straightforward as given in the following pseudo_C code:

```
Level(m)
{
/* step 1: set level to the home level "i" */
level = i;
/* step 2: check for the special case of bucket 0 */
if m == 0 then return(level); /* bucket 0 is always at the home level */
/* step 3: */
Compute : $\beta$, $\gamma_l$, $NS_m$ and $m'$;
/* step 4: determine actual level */
if ($m' \leq NS_m$)
    {
    /* bucket "m" has undergone a split and one should rehash
     * with $OPH_{i+1}$ */
    level = i+1;
    }
return(level);
}
```

4. Insert Routine

The insert routine follows the same concept of the address computation procedure. First, we need to compute the bucket address for the insertion and then append the record to the bucket if space is available. If the addressed bucket is full, for an uncontrolled split mechanism, the routine calls the split procedure and the overflow resolution procedure. The overflow resolution procedure, which is not discussed in this report, would be similar to the CRM method of LH or other overflow bucket allocation scheme.

```
Insert(Key)
{
1. buck_add(Key); /* Compute bucket address of "Key" */
2. "Read bucket";
3. if "empty space available"
        then "insert record";
        else "call split routine and overflow resolution procedure";
4. return;
}
```

The deletion algorithm is similar but would require a merge routine instead of a split routine and is not discussed here.

9.5.6. Inverted Surrogate Files with DRSAM

In this section, we extend the DRSAM technique to Inverted Surrogate files and propose the "Inverted Dynamic Surrogate File" (IDSF) that is meant to replace the static inverted lists and TILs as applied to surrogate files.

The distribution of the value of an attribute over its domain is assumed to be "quasi-uniform", with the additional constraint that the peak value of the value distribution factor $C_i < b$, where b is the number of entries in a "block". This restriction implies that DRSAM files seem to be especially suited for the case when all equal BR values fit into one block and its associated overflow area. This restriction shall be relaxed in the future by providing proper control schemes. It is noted that this assumption is made for any dynamic hashing technique such as linear hashing (LH), extendible hashing (EH) or others. Furthermore, in the case of inverted surrogate lists, this restriction is not overwhelming and would be easily relaxed as the surrogate file records are small in size so that "b" is expected to be relatively large (more than 300 per bucket).

9.5.6.1. System Model

Using proper hashing functions on the attributes of a tuple in a relational table (referring to relational data bases), we can build a surrogate file representation of that table. Figure 9.5.18 shows an example of a surrogate file for a knowledge base relation, with the entries of column Ar_i representing the values for the i-th attribute binary representation (BR_i) in a tuple. For each tuple in the main file and in its surrogate image, we have attached a unique identifier (Uid). This unique identifier could be one that is provided serially or is actually the BR of the "primary key" in the relation. In our discussion we assume that the Uids are serially generated and Figure 9.5.18 is representative of a 4 arguments relation.

Uid	Ar_1	Ar_2	Ar_3	Ar_4
uid1	......	br1	......	......
uid2		br2=010011010		
uid3		br3		
uid4		br1		
uid5		br4=010101011		
uid6		br1		
uid7		br5=010101110		
uid8		br6		
uid9		br6		
uid10		br7		

Figure 9.5.18 A Surrogate Image of a Knowledge Base Relation

A fully inverted dynamic surrogate file (IDSF) would consist of "i" DRSAM files. The DRSAM file records for attribute "i" are composed of the BR of the hashed values (instantiations) for that attribute, with the corresponding Uid. The reader should be aware that, in an actual implementations, only Postfix(BR,(#BR-l)) bits are needed to be attached with the unique identifier, with Postfix(K,n) as the right "n" bits of "K" and "l" the home level of the DRSAM file under consideration. This would mean that the inverted files would more efficiently use the space as the file grows. A typical DRSAM surrogate file block with its associated records is shown in Figures 9.5.19.a and b.

The file is assumed at level l=3. If the block structure of Figure 9.5.19.b is followed, we would be dealing with variable length records depending on the level "i" of the addressed block. This would certainly increase the complexity of the managing software and its associated hardware to deal with such blocking schemes. The pros and cons of such a blocking structure will be investigated as it leads to an efficient use of the storage space for the inverted surrogate lists and implies a lower collision probability as the file expands.

BR	Uid
010011010	uid2
010101011	uid5
010101110	uid7

Figure 9.5.19.a. Block structure with fixed length records.

Postfix(BR,(#BR-l))	Uid
011010	uid2
101011	uid5
101110	uid7

Figure 9.5.19.b. Block structure with variable length records.

9.5.6.2. An Estimate of the Storage Overhead

In this section, we provide an estimate of the storage overhead for inverted surrogate files based on DRSAM. Consider two relation file sizes of 10 Mbytes and 1 Gbytes and assume that each file has six arguments ($A_r = 6$) of 15 characters each. With B = 2 Kbytes, $C_i = 1$, Uids encoded with a 4 bytes word and a file load factor of 0.8, we compute the approximate values of Table 9.5.1.

Meaning	$S_{db} = 10^7$ bytes	$S_{db} = 10^9$ bytes
Minimum number of bits for a BR: $\lceil \log_2 N \rceil$	17	24
Number of records in the relation (N)	11×10^4	11×10^6
Inverted surrogate list size to relation size ratio ($\%S_i$)	8%	8-10%
DRSAM file level (l)	9	16
BR-l (bits)	8	8
Compression ratio of variable to fixed length record formats	0.81	0.71

Table 9.5.1

It is noted that the results are conservative estimates and a more accurate analysis will be provided in the future. The value of BR-l = 8 bits checks with the intuitive feeling that variable length records, as advocated in Figure 9.5.19.b, are efficient as to the storage use of DRSAM files for inverted surrogate file application. The ratio of the variable length to fixed length records is 0.81 and 0.71 for the 10 Mbytes and 1 Gbyte files respectively. This presents a substantial saving of 17 to 30 % on the inverted surrogate list size with fixed record formats. A value of 8% of S_{db}, per inverted list, is a good estimate for a preliminary evaluation of an inverted surrogate list size. In contrast to conventional inverted lists, this preliminary estimate shows that inverted surrogate files do not require an overhead that is in excess of the data base size.

The analysis of TIL files assumes static files (or stable files) that are initially loaded and stored in compact form. For inverted lists built with DRSAM, the storage overhead is larger and is caused by the additional space required to manage volatile files. This overhead is still less than 50 % of the original data base.

9.5.6.3. Query Response Time

In this section we provide a preliminary insight to the equations that govern the query response time for inverted surrogate files. Like TIL files, the query response time (QT) for IDSF is divided into three processes:

1) Surrogate file processing and Uid retrieval (T_{sp}).
2) Uid intersection time (T_{it}).
3) Data base access time (T_{dp}) to read the identified record(s) satisfying the query.

The query response time is written as: $QT = T_{sp} + T_{it} + T_{dp}$

1. Surrogate File Processing Time

T_{sp} is determined by the number of disk accesses required to retrieve the matching Uids. With AC(i) as the average disk access cost for the surrogate inverted list of argument "i", "Q" the query specification with R_q arguments and Ovl(i) the average overflow chain length for the DRSAM file of argument "i", the average surrogate file processing time can be written as:

$$T_{sp} = \sum_{i \in Q} AC(i)$$

$$AC(i) = (1 + Ovl(i)) \times T_d$$

where T_d is the retrieval time of a secondary storage block. We did not account for the search time of the retrieved blocks as it can be overlapped with the retrieval process and is neglected. Furthermore, for a DRSAM file with a load factor of 0.8, AC(i) is expected to be around 1.2 disk accesses (if we assume similar characteristics as for LH).

2. Intersection Time

With no loss of generality, we assume conjunctive queries as the union operation for disjunctive queries has the same level of complexity as the intersection operation. Two cases are considered:

$R_q = 1$: no intersection is required.

$R_q > 1$: when more than one argument value is specified in a query, the lists of retrieved Uids must be intersected. Denoting by $NC(R_q)$, the number of comparisons required to perform the intersection operation, T_W the average word comparison time and WL the word length, the total intersection time is written as:

$$T_{it} = \begin{cases} T_w \times \left\lceil \dfrac{\lceil \log_2 N \rceil}{WL} \right\rceil \times NC(R_q) & \text{if } R_q > 1 \\ 0 & R_q = 1 \end{cases}$$

An estimate of the number of comparison steps, $NC(R_q)$, for the intersection operation is derived in [HAC88, Appendix 2].

3. Data Base Access Time

With GD as the number of good responses to a query and the probability $\left(\frac{1}{\left\lceil \frac{S_{db}}{B} \right\rceil}\right)$ of a given response to be in a specific block, the data base access time is, following Cardenas' equation [CAR75] and assuming direct access to the main data base:

$$T_{dp} = T_d \times \left\lceil \frac{S_{db}}{B} \right\rceil \times \left(1 - \left(1 - \frac{1}{\left\lceil \frac{S_{db}}{B} \right\rceil}\right)^{GD}\right)$$

Following [HAC88, Appendix 2], the number of good responses is estimated as:

$$GD = N \prod_{i \in R_q} \left(\frac{C_i}{N}\right)$$

It is observed that the data base access equation is based on successive selections with replacement. Yao [YAO77] discusses selection without replacement and points out the cases where Cardenas' equation gives rise to a significant error. For our purposes, Cardenas' approach is satisfactory as the number of good responses is expected to be small for very large knowledge bases.

9.5.7. Parallel Back End Architecture for IDSF

In this section, we describe and analyze the benefits of a parallel back end architecture for the management of knowledge based systems with inverted surrogate files.

9.5.7.1. Back End System

Shown in Figure 9.5.20 is a back end system for the management of a very large extensional data base of facts. This system will also manage many intentional data bases (sets of inference rules), but those are not shown on the diagram. We assume that there are many gigabytes of fact data stored on the EDB disks. Likewise, there are several gigabytes of surrogate file data stored on the SF disks (SFD). Since the relational model is assumed, the facts are stored by relation and then by tuple unique identifier within relations. We will access the EDB only by relation name and then by tuple identifier, so a dynamic hashing method that minimizes disk accesses can be used, one of them being specifically DRSAM as presented.

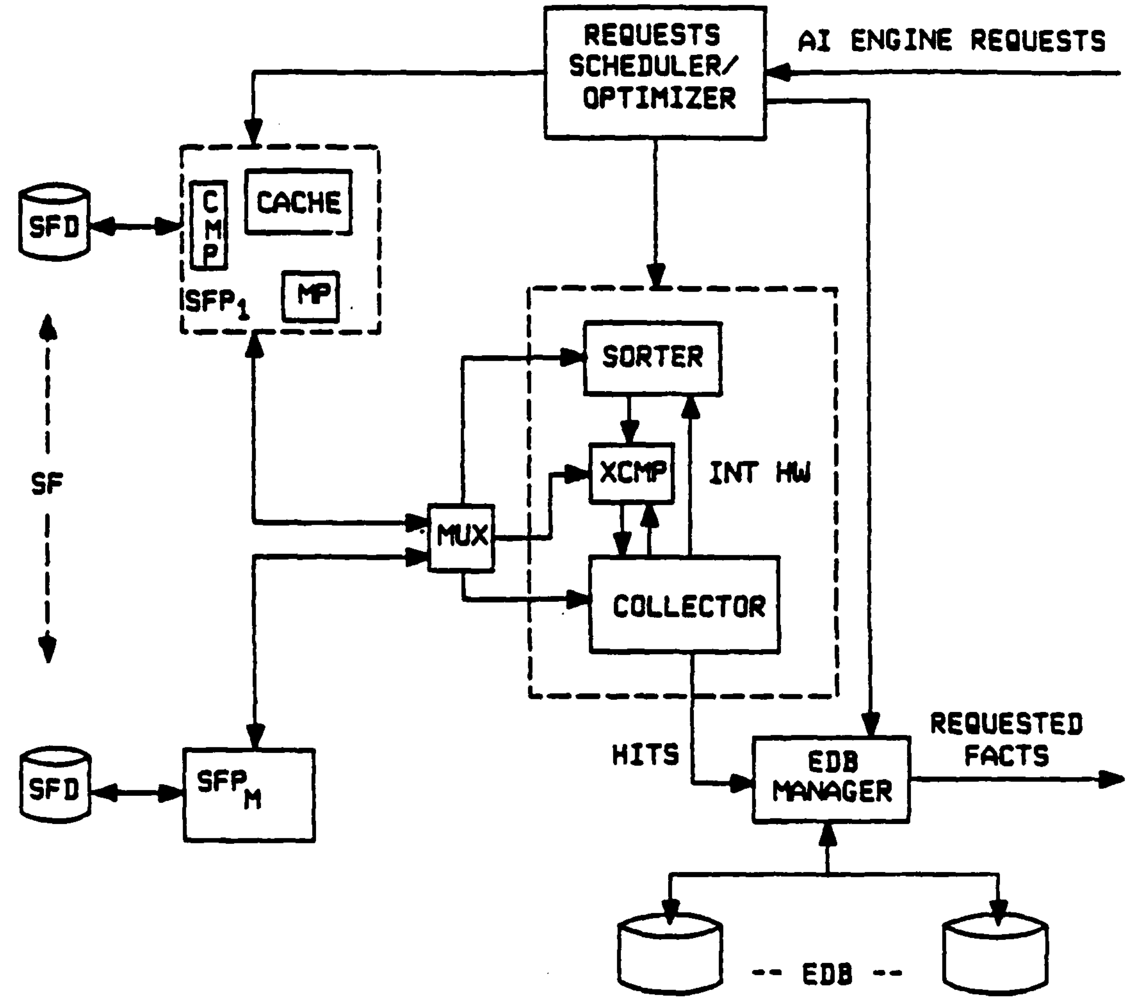


Figure 9.5.20 BACK END ARCHITECTURE FOR FACT MANAGEMENT.

As an example, assume that a user's request requires access to only two lists. The relevant block(s) from the first list would be retrieved from the SFD and input to its associated surrogate file processor (SFP) where on the fly comparisons are made for matches by the comparator (CMP). Note that the SFP consists of a comparator (CMP) and cache (CACHE) with their associated control microprocessor (MP). The unique identifiers would be stripped off and sent to the Intersector Hardware block (INT HW) through the multiplexer (MUX). The list of Uids is piped in the pipeline sorter (SORTER) and then fed to the cross-lists comparator (XCMP).

Meanwhile, the second list is processed in a similar way and sent to the XCMP module. Then, the two resulting lists of possible responses are intersected by the XCMP block. The output of Uids (if any) is sent to the collector (COLLECTOR) that acts as a buffer and the block of good responses (HITS) is passed on to the Extensional Data Base Manager (EDBM) for processing. The EDBM will retrieve the facts, compare them with the search criteria to insure that a collision has not occurred, put them in blocks, and sends them to the logic programming engine.

In the case where more than two lists are to be intersected, the outcome of the two lists intersection is fed back from the COLLECTOR to the XCMP block for a new cross comparison operation with the third list coming from the SFD/SFP pairs. This process is continued until all the arguments in the query are properly processed. When a single argument query is considered, the MUX passes the incoming list from the SFD/SFP pair to the COLLECTOR that relays it to the EDB manager. The complete system can be viewed as a three level pipeline controlled by the Requests Scheduler/Optimizer.

9.5.7.2. Analysis of the Proposed Architecture

In this section, we analyze the motivations and the benefits of the described architecture. One recurrent criticism against the use of inverted file structures is that their performance degrades as the number of arguments in a query increases. A good algorithm would tend to perform in the opposite way, as one hopes to do work proportional to the expected number of tuples in an answer. This criticism is assessed based on the sequential processing of the surrogate inverted lists, but is mitigated if parallel processing algorithms running on multi-processor architectures are designed for transformed inverted lists. We will have to look at the equations for the different components of the query response time (QT), namely T_{sp}, T_{it} and T_{dp}.

1. Surrogate Files Processing Speedup

We observe that T_{sp} is proportional to the number of arguments in a query (R_q) and is related to the disk access cost for the retrieval of the inverted lists indices. The IDSF structure is well suited for parallel processing through the distribution of the inverted lists to multiple storage and associated processor units (SFP). For the case of a single user* queries on a relation with degree "d", an O(d) speedup for the surrogate files processing time can be achieved with a maximum of "d" SFD/SFP pairs. For a multi-user system, the speedup which can be

* A "user" is referred to as the application programmer. A single user refers to a single application environment versus a multi-user i.e multiple applications environment.

achieved is a function of the number of SFD/SFP pairs and the application being considered. The surrogate file will actually consist of many sets of inverted subfiles, one set for each relation. Those sets will be distributed over the SF disks in order to insure maximum parallelism in disk accessing.

The distribution algorithm follows an optimization criterion related to the application on hand. We note that the assignment problem is NP_Complete and heuristic algorithms, specifically designed for the proposed architecture, are being presently developed for the proper distribution of the surrogate inverted lists. The outcome of the optimization algorithm would be a storage mapping of the surrogate inverted lists that is used by the Requests Scheduler/Optimizer for query optimization.

For a distributed storage allocation system, the equation for the surrogate file processing time should be modified to account for the access of the lookup tables that are bound to exist. The use of cache memory (CACHE) in each SFP unit is to store the lookup tables that are small in size.

2. Intersection Operation Speedup

The analysis for the intersection operation cost [HAC88] shows that T_{it} heavily depends on C_i: while acceptable for small data bases, T_{it} becomes a computation bottleneck for medium and large data/knowledge bases with high average redundancy factors (C_i). With a VLDKB, the analysis reflects an essential need for special intersection hardware (referred to as the Intersector) to cope with the computation intensive intersection operation. In Figure 9.5.20, the Intersector is part of the INT HW block and mainly consists of the pipeline sorter (SORTER) and cross-list comparator (XCMP) units. The sorter is essential and shall be optimized to handle large lists of Uids as they present the computation bottleneck of the intersection operation. The XCMP block is used to cross compare the sorted list of Uids from the output of the SORTER with an incoming list of Uids from a SFP.

With L_{min} as the minimum length of the lists involved in the intersection operation, an $O(L_{min})$ computation steps could be achieved with the Intersector. Compared with an $O(L_{min} \times \log_2 L_{min})$ computation steps of the best sequential algorithm, the speedup achieved with the hardware Intersector would be $O(\log_2 L_{min})$.

For high query rates, the operation of the INT HW block and the SFD/SFPs are overlapped, thus increasing the throughput of the system. The number of Intersector blocks is not bound to one, as shown in Figure 9.5.20, and is a function of the throughput constraint of the design. Maximizing the level of pipelining between the SFD/SFP pairs and the INT HW block(s) is an additional requirement on the optimization algorithm. It is worth noting that a different intersection hardware could be derived based on a parallel cartesian product algorithm. We believe that such hardware would be more elaborate than the sorter/cross comparator combination.

3. Comments on the Data Base Access Time

Data base access time (T_{dp}) depends on the locality of the good responses and would be determined by the clustering scheme for the tuples in the existing EDB. In the analysis, T_{dp} is derived following Cardenas' assumptions [CAR75] of uniform distribution for the records over the EDB secondary storage blocks. In a multi-user environment, clustering can achieve optimal T_{dp} values for one user while degrading the response time for another. EDB clustering is an open design problem that lies in the class of NP_Complete problems.

9.5.8. Open Research Problems and Future work

One unusual phenomenon common to all dynamic schemes and therefore expected with DRSAM and IDSF is the following: if a collision occurs, it may happen that splitting one level only would move all the data into one block instead of dividing it onto the 2 buckets. Let us assume that the hashing depth is x bits, then the splitting function resolves it by dividing the information in two sets that differ through the (x+1)th bit. It could be the case that all the data in the bucket does not differ through this bit but through a higher level bit, then the split results into an empty bucket and a full bucket with the possibility that an overflow record is attached to it. This means that if the attribute values distribution is highly non uniform, LH, EH and also DRSAM may result into a file structure with long overflow chains and low load factor. Controlled splitting is used to set the load factor as required.

Different control mechanisms could be added to alleviate this behavior. One of them would be to have multiple levels existing concurrently during the expansion cycle of the file. While the present techniques are based on a two level scheme, we believe that such multi-level schemes could be adapted, especially within a fully distributed environment. In this case, a table lookup is provided and with a small additional storage overhead, more than one expansion level could exist at a given time. The idea is to partition the file into quanta that are expanded independently as required, following closely the distribution of the key values.

Another promising approach was studied by Larson [LAR79] in a different context. He analyzed the use of "repeated hashing" as a technique to handle overflows and concluded that the usefulness of this technique is doubtful. We believe that the pessimistic results reported by Larson are due to the fact that deletions, in his analysis, are handled by marking the deleted records. This would mean that repeated hashing would behave in a similar way as usual overflow control techniques. In the case of dynamic hashing, repeated hashing could become an interesting and simple method to extend dynamic hashing methods to handle overflows as well. Dynamic hashing schemes, like DRSAM, handle deletions and insertions by natural "contraction" and "expansion". The problem of having a file with marked "unuseful data" is avoided and repeated hashing would be a natural extension to DRSAM files.

Ramamohanarao et al [RAM84] analyzed this idea as applied for linea.: hashing and derived a general scheme referred to as "recursive linear hashing". This method seems promising and is presently being investigated for DRSAM.

DRSAM (and IDSF) present a promising alternative to ISAM, VSAM (and TIL) with overflow buckets. Those structures are expected to show a near perfect retrieval cost for random access while they preserve (within some restrictions) the ordered characteristic of TIL. The major asset of IDSF with respect to TIL being their efficiency and ease of maintenance when applied to volatile files. Furthemore, knowledge about the distribution function would help the data base designer fine tune the IDSF structure to the application on hand. The load factor of IDSF is expected to be comparable to the one of LH and other extensions techniques to LH (like partial expansion [LAR80] and multidimensional designs [OUK83]) could be applied as well to IDSF. Partial expansions and controlled splitting techniques should result in a high load factor (around 0.9) with little degradation in performance.

DRSAM (or IDSF) is a promising dynamic file management technique. We mainly discussed its application in the context of operating systems that handle contiguous file allocation schemes. In forthcoming work, we will analyze it within the general distributed allocation scheme and in this case, as for LH, a lookup table is necessary. The concept of quantified allocation will be studied and applied and we will show that with minimal additional storage overhead, the file structure can be made to expand randomly and would adapt to almost any distribution function for the values of an argument over its domain. This would tend to minimize the overflow chains and thus decrease the oscillation in response time detected for LH. Split control and partial expansions will be studied as well and overflow handling will be analyzed with the usual overflow chaining, the repeated hashing or other suitable overflow handling mechanism.

Based on IDSF structures, we introduced a parallel architecture for a Very Large Data/Knowledge Base. We intend to carry a detailed analysis and development of this architecture. Wherever needed, analytical as well as computer simulated models will be derived.

9.6 Management of Very Large Rule Bases

Presented in this section are techniques for managing a very large rule base to support diverse requirements of parallel logic programming systems based on surrogate files and associative processors. Future work on general rule indexing schemes are described in section 9.6.4.

9.6.1 Parallel Execution of Logic Programming

Conery [CON87] has classified the inherent parallelism in logic programming systems into three major categories: AND-Parallelism, OR-Parallelism and Low-level Parallelism. Our major concern here is a special case of OR-parallelism called search parallelism which has been defined as a parallel distributed search to find every clause with a head that unifies with the selected goal. Since a search performed by integrated knowledge base machines should be based on unification rather than equality, it is well known that an effi^ient implementation of unification is the central issue in logic based systems. Several processors dedicated to the unification operation have been proposed in recent years to accelerate this most time-consuming operation in logic programming evaluation [WOO85] [SHO86] [STO86].

Informally, the main purpose of unification is to make two or more terms identical by proper and the most general substitutions for logical variables in the terms. A term is defined as follows [LLO84]:

(1) A variable is a term denoted by a capital letter such as X,Y,Z,...

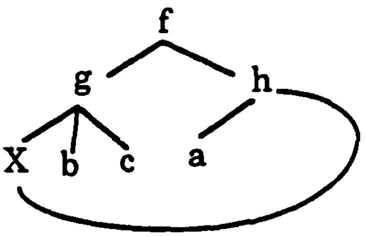
(2) A constant is a term denoted by a lower case letter such as a,b,..

(3) If f is an n-ary function and $t_1,..., t_n$ are terms, then $f(t_1,..,t_n)$ is a term.

Ever since Robinson introduced the basic algorithm of the unification operation for the resolution principle [ROB65], more efficient algorithms have been proposed and the complexity of the unification operation has been analyzed by many researchers [DWO84] [VIT86]. Among them, two algorithms [PAT78] [MAR82] are claimed to be linear. These algorithms are based on a complex data structure called Directed Acyclic Graph (DAG). Also, Morita proposed a linear representation of a term suited to stream processing of unification [MOR86]. The DAG and linear representations of a term are shown in Figure 9.6.1 (a) and (b) respectively.

Our major concern in implementing unification for very large rule bases is finding all potential candidate clauses within a small amount of time so that we can deal with real time applications. Since the full unification on such data will require a heavy processing load, our goal may not be achieved without restricting unification. Furthermore, the results of [DWO84] indicate that, since unification is inherently sequential, even parallel evaluation of a unification algorithm may not offer a considerable speed-up over a sequential one.

The major processing load stems from 'occur checks' to prevent the unification from entering an infinite loop. That is, when testing if a variable X unifies with a structured term t, a check should be done whether X occurs in t (i.e. {X/f(X)}). We can eliminate these requirements by adopting mode declarations to construct a 'standard form' of clauses as in PARLOG [CLA86] where the structured arguments appearing in clause heads can be transferred to the bodies of clauses.



(a) DAG

(f2)(g3)(X0)(b0)(c0)(h2)(a0)(X0)

(b) Charcter String

Figure 9.6.1 The Representation of a Term (f(g(X,b,c), h(a,X))

A PARLOG program that possesses a single solution consists of a sequence of guarded Horn clauses. A guarded Horn clause of PARLOG has the form

A:-$G_1,G_2,..,G_m$:$B_1,B_2,..,B_n$.

$m,n \geq 0$

If m=0 then the commit operator can be omitted. A candidate clause of PARLOG is one which succeeds in all input matching with the call (subquery) and whose guard literals ($G_1,G_2,..,G_m$) are proven to be true. PARLOG exploits "mode" declarations for the clauses in the single solution relation to avoid the requirement of full unification, and to control process synchronization [CLA86]. A mode declaration for a predicate can constrain the unification between a goal and a clause (head) in a program. Mode declaration is of the form

mode $R(m_1,m_2,....,m_k)$

where R is a predicate name and each m_i is either '?' or '^'.

An argument annotated with a '?' in the mode declaration for a predicate can only be used for input matching against the corresponding argument of a call. That is, the unification between a call and the head of the clause is successful only if the corresponding argument in the call is instantiated (i.e. not a variable). Otherwise the evaluation suspends. On the other hand, an argument annotated with a '^' must be used for output matching against a variable of the corresponding position of a call. In other words, the corresponding argument of a call should be an uninstantiated variable on unification. If the argument is not an uninstantiated variable, the unification fails.

The mode declaration is used to determine the "standard form" of clauses at the first stage of compilation. In the standard form, all complex terms appearing in the heads of clauses can be represented as pure variables, and all input and output matching between a call and the heads of clauses are translated to explicit unification primitives instead of general unification.

Consider, for example, a simple PARLOG program

```
mode member(?,?).
member( H,[H|T] ).
member( H,[X|T] ) :- ~H=X : member(H,T).
```

where ':' is the commit operator and ~H=X is a guard.

This program can be mapped into the standard form

```
member(H,Y) :- [X|T]<=Y,H=X:.
member(H,Y) :- [X|T]<=Y,~H=X: member(H,T).
```

The term [X| T] that was in the second argument position of the second clause head appears as [X|T]<=Y because it has the mode '?'. Here '<=' is the one way unification primitive that can only bind variables in its left argument([X| T]). This implies that this term can only be used for input matching against the given argument Y of the call. The repeated use of the term H in the head of the first clause is detected as an implicit test because both terms have the mode '?'. Thus the term [H| T] is changed to [X| T] (here X is an arbitrary variable) and an explicit test unification primitive '=' is added in the guard. In order to change a non-variable term with the mode '^' to the standard form, the assignment unification primitive ':=' should be used in the body. The unification primitives of PARLOG are described in [CLA86]. Maluszynski and Komorowski [MAL85] have also discussed the use of mode to constrain full unification.

Consequently, the structured arguments (e.g. [H| T]) in the clause head can be transferred to the guard or body of a clause as shown in the above examples.

9.6.2 Rule Indexing Schemes for Surrogate Files: CCW-1

In previous sections, we have shown the use of surrogate files for partial match retrieval on large sets of facts with varying degrees and cardinalities. In retrieving facts, we assume that the facts are stored in such a way that one first accesses the relation and then a particular tuple using a unique identifier. Thus, we do not need to transform the predicate name (e.g. parent) for the facts. We obtain the unique identifier from processing the surrogate file, and the name of the relation from the given query. Thus, the storage structure for the facts themselves would be very simple and the desired facts can be retrieved in at most two disk accesses. Most relational operations such as selection and join, which are required for the bottom-up query processing in logic-oriented database systems, can be performed on the surrogate file rather than on the actual database. This makes relational operations much faster and increases the system's performance when a large volume of ground facts exist.

In a CCW representation of a clause head, we don't consider structured terms. The clause head contains pure variables and constants as arguments by the transformation technique adopting the mode declaration. General rule indexing schemes are considered in section 9.6.4.

Variables should be distinguished from constants. This can be done by setting the tag bit (most significant bit) of the CCW to ' 1 '. Unlike facts, there are only a small number of rules that define a predicate, i.e. rules with the same head.

Thus, we need to transform the predicate name as well as arguments.

Suppose we have rules for 'ancestor',

ancestor(X,Y):- parent(X,Z),ancestor(Z,Y).
ancestor(X,Y):- parent(X,Y).

We hash the predicate name and arguments by the same hashing function used in CCW for facts. The number of arguments is also concatenated to the hashed value of predicate name.

H(ancestor 2)	H(X)	H(Y)
|	|	|
011100010	100100111	100101001.

The CCW representations for the two rules would be the same except for the uid's to be attached to them.

```
011100010 | 100100111 | 100101001 |uid_1
011100010 | 100100111 | 100101001 |uid_2
```

Thus, a surrogate file can be used to find the corresponding bodies of clauses with which a goal can unify via uid's.

This method guarantees retrieval of all desired terms (clause heads or facts) although, due to possible collisions resulting from the hashing method some undesired terms may be retrieved. A longer word length for the CCW can minimize such collisions, and post retrieval comparisons can be used to eliminate unwanted terms.

In the next section, we describe how one might perform partial unification on a surrogate file by proposing a special associative memory for bidirectional don't care matches.

9.6.3 Partial Unification on Surrogate Files

In this section, we present the basic idea of unification on a surrogate file using an associative processor. We have shown in section 9.6.1 how to transfer the complex structured arguments in the head of a clause to its body. For simplicity, we assume that the query contains only pure variables and constants. Thus, the query code word (QCW) can be encoded by the same technique as described in section 9.6.2.

First, for all constants in a QCW, the corresponding arguments of the CCW must be either the same constant or a variable in order for the terms to be unifiable (Input matching condition).

In the input matching step, we regard all variables as "don't care match" indicators. Unlike usual "don't care" matches, however, we need bidirectional don't care matches because the data residing in associative memory, as well as the QCW, may also contain variables. Since general associative memories do not provide this capability, a special associative memory is required. We designed an enhanced associative memory for bidirectional don't care matches, as shown in Figure 9.6.2. Since by assumption only variables and constants appear in a QCW, input matching among a QCW and a number of CCW's, each representing a head of a clause, can be performed in O(1) time* (i.e. constant time).

By input matching, most unqualified terms can be pruned. After input matching, we assume that the qualified terms (heads) are read one by one for further processing. Thus post processing will be required for only a relatively small number of terms, namely the qualified terms.

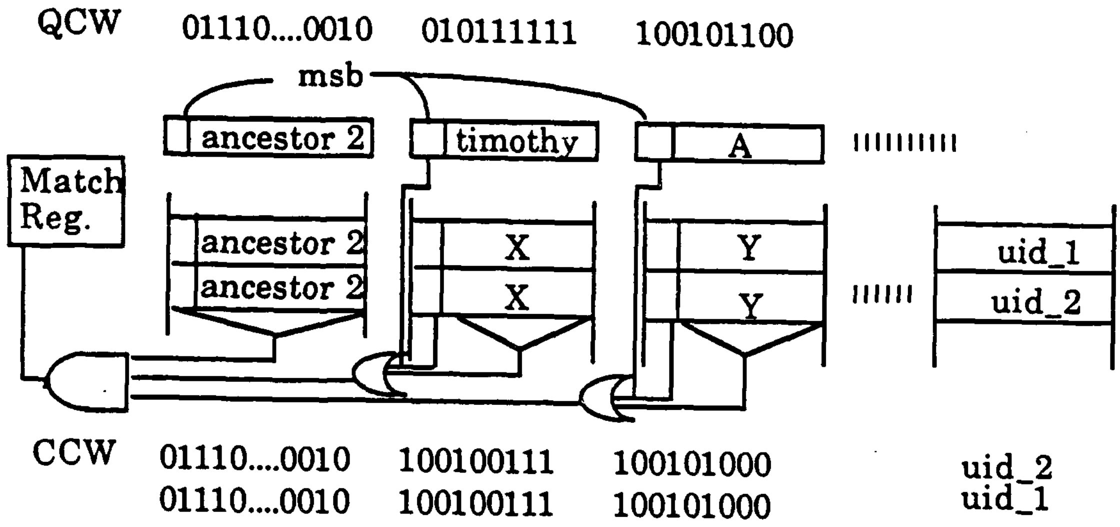


Figure 9.6.2 An Associative Memory for CCW-1

Obviously, the above condition is not sufficient. Consider, for example, two terms of the form q(a,X,b) and q(Y,a,Y). Though they satisfy the condition, they are not unifiable. We need post processing for the shared variables that appear in arguments of qualified CCW's. If the same variable appears in arguments of a CCW, they should be bound to the same constant or variable (Input matching consistency).

* To process a QCW with a longer word length than that of the associative memory's, the QCW should be split into parts and the unification performed on the parts in sequence. Since the unification for the QCW has to be performed in parts, the variable bindings along with the content of match registers resulting form that unification should be stored for the next unification. For simplicity, we assume that the word size of a QCW is always shorter than that of the associative memory.

The prime objective of unification is to find proper bindings for variables. After input matching and consistency checking are perfnrmed, the variables of qualified terms (CCW's) are substituted by the constants obtained from input matching. The reverse operation is required to bind variables in QCW. If these terms are unifiable, then the similar condition as the input matching condition will be satisfied. That is, for all constants in a qualified CCW, the corresponding arguments of QCW should be either the same constants or variables (Output matching condition).

Finally, a consistency check for the variables in the QCW needs to be performed. That is, if the same variables appear in the arguments of the QCW, they should be bound to the same constant or variable (Output matching consistency). The unification method always works with the function-free terms. An algorithm for parallel evaluation of logic programs and some considerations on its hardware realization have been discussed in [SHI87].

9.6.4 Future Work on Managing Very Large Rule Bases

The rule indexing method described in section 9.6.2, CCW-1, provides an efficient mechanism in searching possible candidate clauses as well as in detecting binding conflicts among shared variables in early stage of execution. However, since this scheme is based on guarded Horn clauses and mode declarations, its application is somewhat limited to the parallel logic programming paradigm.

We are currently developing enhanced rule indexing methods to provide more general and efficient accessing mechanism to VLKBs consisting of First Order Logic clauses. Those schemes currently investigated are featuring type-checking augmentation of CCW-1 (CCW-2), a CCW scheme for general terms (CCW-3), and a CCW scheme for general clauses (CCW-4).

CCW-2 has a similar structure to CCW-1, which can be constructed by concatenating transformed code words obtained from the arguments along with the hashed value of a predicate name. Each code word for an argument is divided into two fields; tag field and value field. Unlike CCW-1, however, the tag field can represent any argument types including lists and structured terms as well as variables and constants. The value field contains the transformed representation of the corresponding argument according to the contents of its tag field. For example, if a tag indicates the argument type of structured term, then the value field contains the hashed value of primary functor, while if a tag is for a variable argument, the value filed represents variable identification number. This scheme can be viewed as an augmentation of CCW-1 with the indexing scheme used in Warren's Abstract Prolog Instruction Set [WAR83]. However, in [WAR83], only the first argument is indexed. Table 9.6.1 shows an initial design of CCW-2 scheme. In contrast to CCW-1, CCW-2 can be used for current Prolog systems and does not require mode declarations. It is expected that false drop can be considerably reduced when compared to previously proposed schemes such as [WIS84] [RAM86] [COL86] [SHI87] [WAD87] without sacrificing the compactness and uniformity of CCW.

Argument Type	Tag Field	Contents of Value Field
Constant	00x	Hashed Value of the constant
Function	01x	Hashed Value of the Primary Functor/Arity
List	100	Hashed Value of the CAR constant
	101	Variable ID for the CAR variable
Variable	11x	Variable ID

Table 9.6.1 Code Word Description in CCW-2

Extending the rule indexing schemes to arbitrary complex terms is one of the most attractive research topic for the next year regarding rule indexing schemes since we can perform almost exact unification on surrogate files and it is more efficient than performing unification on actual terms. General term indexing can be achieved by using a term descriptor (TD) in the position of a structured term or list. In addition, this indexing scheme can be used as a basis for processing more complex EDBs having structured terms as arguments.

Indexing clauses to perform resolution procedures on surrogate files (CCW-4) will be also investigated. A code for CCW-4 can be obtained by generating CCW-3 codes for all predicates in a clause together with a clause descriptor containing pointers for the body literals. However, this scheme may cause false drop propagation problems during the resolution procedures. That is, after a goal is resolved to a number of subgoals, it would be difficult to detect false drops.

As rule indexing schemes become more general, specialized architectures such as associative memories are less practical. Instead, general purpose parallel processors seem to be adequate for surrogate file processing, especially when a significant portion of the surrogate file can reside in main memory. An initial study revels that massively parallel computer systems with large main memory and simple general purpose processing element could have good performance in processing surrogate files due to the uniform structure of surrogate files. The Connection Machine with 32K processors is currently being used to study various aspects of surrogate files and will also be used for testing various rule indexing schemes.

9.7. Optics in Very Large Knowledge Bases

Optical computing (especially in its analog form) has been widely used in applications like optical image processing, pattern recognition and signal processing due to its highly parallel nature. Another area that can benefit significantly from the advances in optical technology is that of the Very Large Knowledge Bases (VLKB). Optics can play a key role in the future VLKB providing larger storage capacity, higher transfer rates and parallel data manipulation. This section discusses some of the possible improvements in the VLKB performance if optical computing is involved.

9.7.1. The Potential of Optical Computing

The application of optical computing to specific areas should take into account the idiosyncrasies of the problems in the specialized architectures employed in those areas. Some problems may be simplified when a narrower view is taken. In knowledge and data base applications for instance, selection, projection and join are common processing chores. Search of fixed format data (e.g. indices or pointers) could make effective use of optical content-addressable memory which can be implemented by multiplexing a large number of holograms in a thick recording material like lithium niobate [GAY85].

The need for large capacity and high bandwidth secondary storage will probably be satisfied by using optical disks. Optical preprocessing of the retrieved data, without intermediate electrical conversion, will help deal with the extreme data rates. Currently, access times of optical disks are larger than those of magnetic disks. The reason is that the focusing optics are bulkier than the 'flying' miniature heads of magnetic disks. Data rates are comparable, with potential for improvement since optical disk technology is relatively new.

However, in contrast with magnetic media, there are two promising possibilities for increased optical disk performance by at least two orders of magnitude both in terms of access time and sustained data rates. First, the read/write beam could be deflected from track to track very rapidly (on the order of 100 microseconds) by entirely optical means. Second, due to the non-interference of light beams and the relatively large head to medium spacing one could imagine multiple beams being used for reading data with a single head carriage assembly [CAR84]. Alternatively, an unfocused beam could simultaneously read data from more than one point of a transmissive disk surface [MOS87]. This, coupled with the possibility of multiple heads will allow for enormous data rates. If we assume achievement of access times of 100 microseconds and data rates of 300 MBytes/sec, this represents almost two orders of magnitude improvement over current magnetic disks.

Input/Output systems will have to be designed with these rates in mind. Current electronics would be hard pressed to handle them. However, if data could be preprocessed "on the fly" in its optical form, then the ultimate data rate to the electronics would be much lower on the average and the data much "richer" in information. Intelligent use of optical pattern matching could provide us with a set of primitive operations that could help efficiently implement higher order functionality like, for instance, a subset of relational algebra operators.

For applications which demand fast searching of many megabytes of data all this is very promising. But with current electronics technology if every subsystem of a machine needs to cater to such high rates then its cost will be much higher than necessary.

9.7.2. Optical Data/Knowledge Base Machines

Assuming a Data/Knowledge Base Machine (D/KBM) with multiple storage units, multiple processors and the appropriate interconnection network, (Fig. 9.7.1), operating as a back-end machine to a host, we may consider four different implementations involving optical devices.

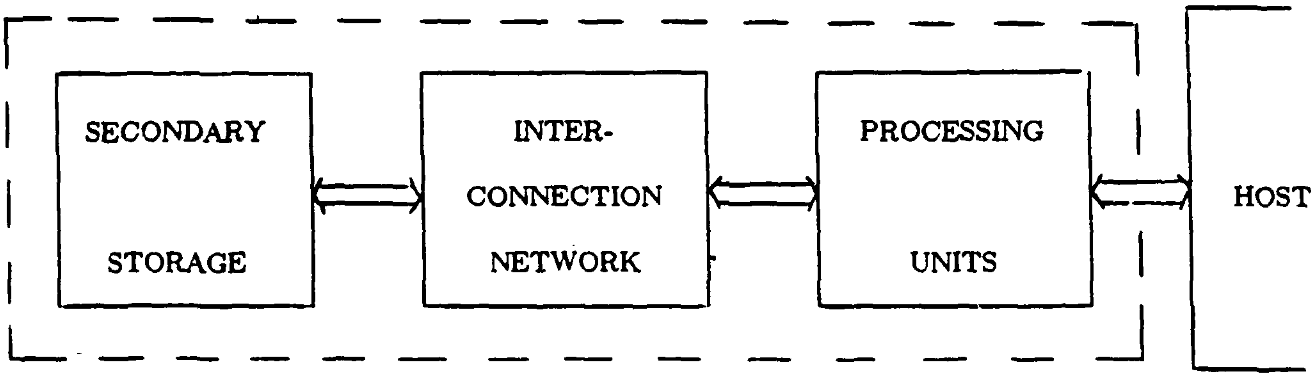


Figure 9.7.1 A back-end Knowledge Base Machine.

a). Optical-Electronic-Electronic, where secondary storage consists of optical disks while the interconnection network and the processors are electronic. This approach suffers from low optical disk transfer rates but may benefit from the availability of the technology. In any case, the only improvement will be t..e capacity increase.

b). Electronic-Optical-Electronic, where secondary memory consists of conventional magnetic disks, data processing is electronic, while the interconnection network will be optical. This approach will theoretically improve the overall sustained I/O rate and allow all kinds of interconnections between disks and processors without conflicts. However, it still retains the bulky magnetic disk units and all the problems associated with them.

c). Optical-Optical-Electronic. Here we have again optical disks for the mass storage. Data in the form of light beams is extracted from the disks, passed through all-optical interconnection network and only when it reaches the electronic processing units is converted to electrical signals. This design takes full advantage of the huge capacity of otical storage and avoids the electron-to-photon conversion, necessary with magnetic disks. The performance of this system will be dramatically improved when multi-beam read/write operations (to be discussed later) become available. However, the anticipated hundreds of MBytes/sec I/O bandwidth from a single disk may drive the electronic processors into saturation, moving the bottleneck to the other end of the machine. For this reason, the fourth approach appears to be the best solution.

d). All-Optical D/KBM. Data is stored, retrieved, transferred and processed completely optically and only when it is sent to the front-end computer, may be converted to electrical signals. The feasibility of such a system is still under dispute because of the currently inferior (compared to the electronic) performance of optical processing techniques. However, a number of key factors, though still experimental ideas, are in favor of this approach and their implementation will signal the "green light" for all-optical information processing.

9.7.3. A Hybrid Opto-Electronic Preprocessor

In this section we discuss the design of a hybrid opto-electronic preprocessor that can help reduce the data rate to the electronics by executing a limited set of functions on the optical data [BER87b].

Figures 9.7.2 and 9.7.3 sketch our hybrid.

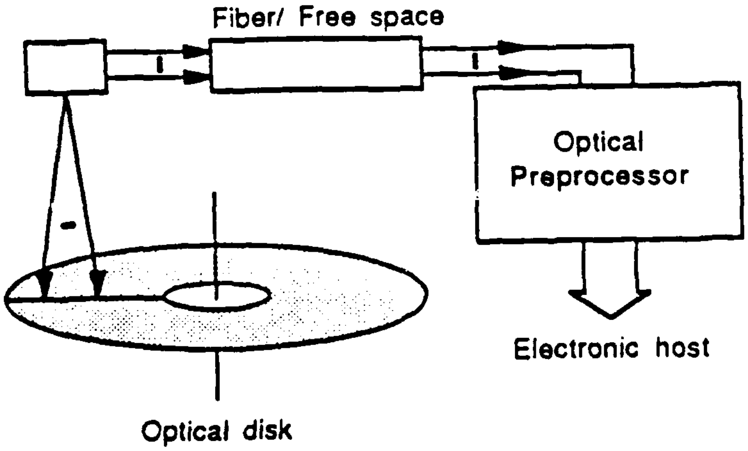


Figure 9.7.2 Optical communication and processing of high data rate disk output.

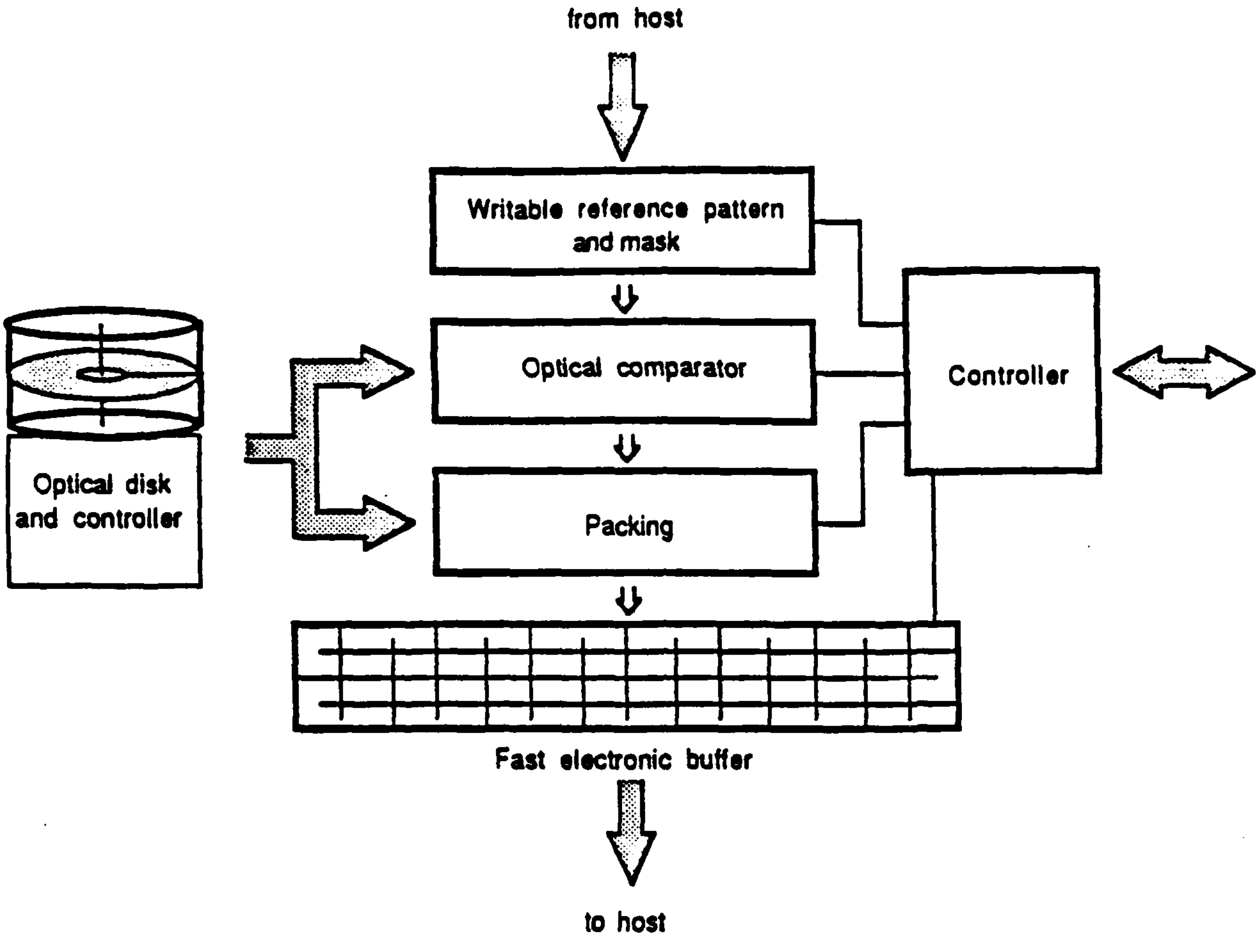


Figure 9.7.3 Block diagram of a hybrid opto-electronic preprocessor.

The optical comparator receives the error-corrected optical bit stream, w-bits wide, from the disk. The bit stream is compared on the fly against an optically-encoded reference pattern. This pattern can contain "don't cares". When the current frame matches the reference pattern, the "interesting" portion of the current frame is latched in a large electronic buffer (two-port cache) which holds it until the host is ready to process it.

The buffer can be implemented as a ring to avoid any internal copying of data. If the buffer ever becomes full, the controller stops the procedure. In this way data rates in the order of 300 MBytes/sec can be accepted and filtered data can be output on demand at a much lower rate.

The w bits of every symbol are encoded in a dual-rail manner by also including the complement of each bit. Two symbols A and B are equal if $A\bar{B}+\bar{A}B=0$ in a bit wise operation. The AND operation is done optically by sequentially propagating a ray of light through two or more points and the OR by imaging two or more rays on the same point [GUI86].

Figure 9.7.4 depicts the flow of data through the process. It consists of the following steps:

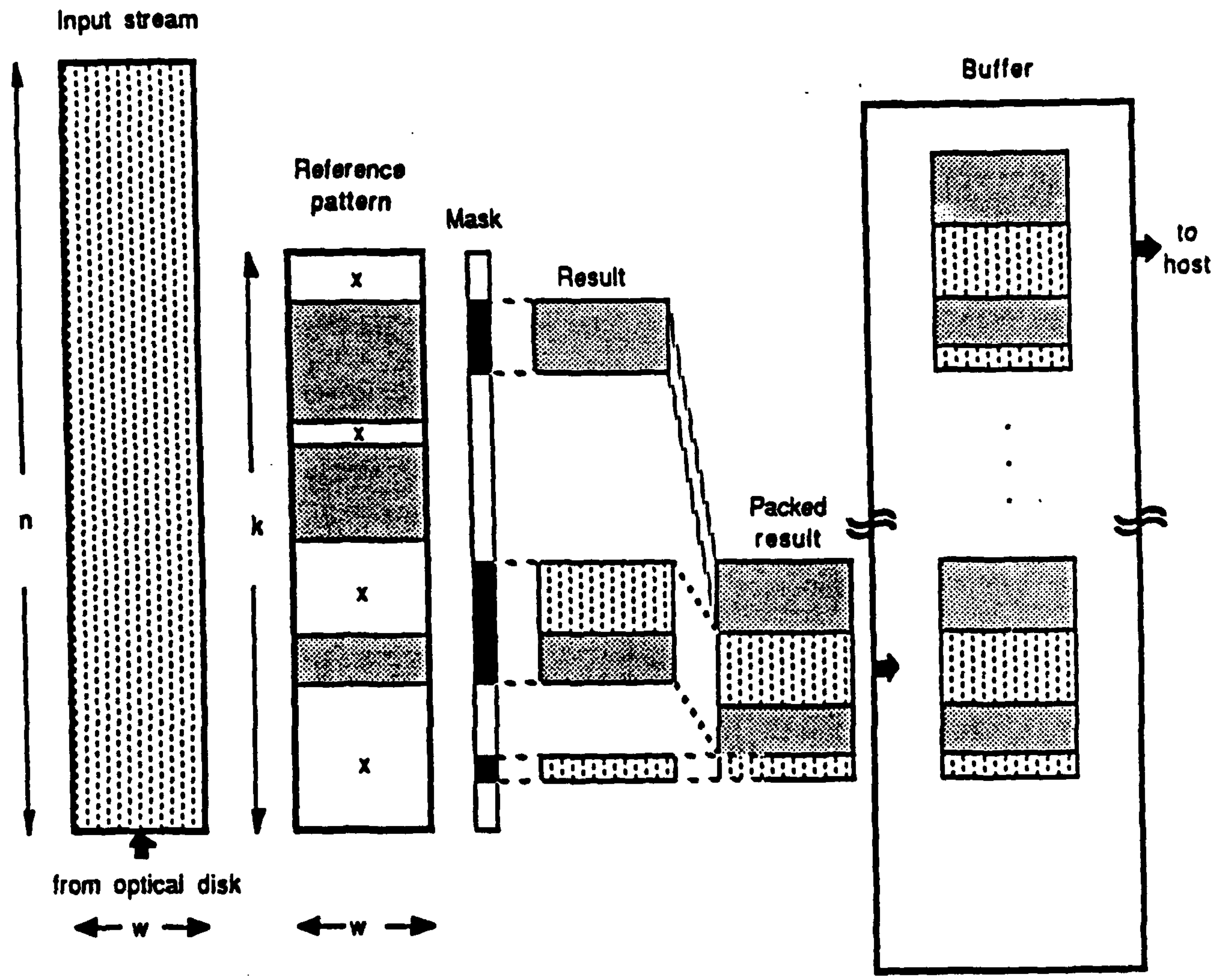


Figure 9.7.4 Processing sequence. Input data from the optical disk is filtered and the results are placed in a large, fast electronic buffer.

1). A w-bits wide stream of data from the optical disk is compared continuously against a reference pattern. The reference pattern may include "don't cares" which are represented as a pair of zeroes in accordance with the dual-rail encoding. The maximum length that can be matched is n symbols.

2). A match occurs if the OR results are all zero for a length of k symbols, where k is the length of the reference pattern, one of the setup parameters of the buffering operation.

3). A mask specifies which parts of the input stream are of interest and the spatially separated parts are "packed" in order to became contiguous.

4). The packed result is transformed to electric signals and stored in the fast electronic cache before the next match occurs.

One way that the matching operation can be implemented is shown in Figure 9.7.5.

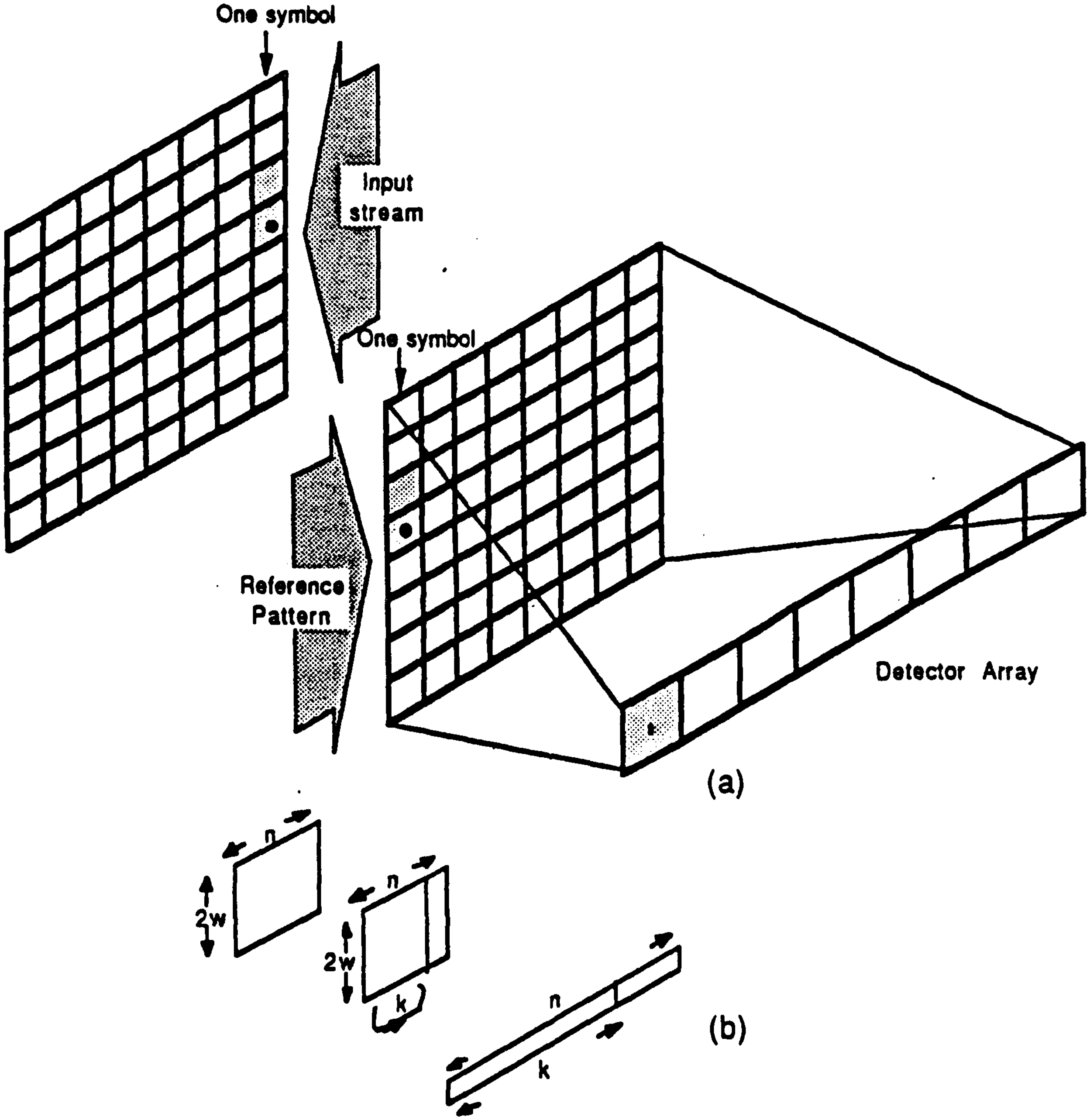


Figure 9.7.5 Three-dimensional arrangement of the optical symbol matcher (after Guilffoyle [GUIL86]). (a) Each bit of every symbol is represented by two complementary light values that are AND-ed with the corresponding bits of the reference pattern. (b) The reference pattern is circulated up to a length of k symbols. A match is detected when the first k detectors register a zero.

The input data stream and the reference pattern stream enter multi-channel acoustooptic cells from two opposing sides. Light beams are imaged in such a way that the complemented bits of the input stream symbols are AND-ed with the uncomplemented bits of the reference pattern stream symbols (and vice versa), bit per bit, symbol per symbol. Each detector accepts light from the 2w positions of every symbol (OR operation). When the output is zero on the first k of the detectors then a match has been detected. The operation depends on the circulation of a pattern of length $k \leq n$ symbols in the optical device that is driven by the reference pattern.

When a match is detected the interesting portion of the input pattern (according to the mask) is packed and kept in the buffer. Packing entails applying a position-dependent amount of delay to predefined regions of the input pattern while it propagates. Hence, it should not be very difficult to implement. Finally, the contents of the buffer can be accessed and updated by means of a few, simple electronic counters.

In terms of relational algebra operators the preprocessor we have outlined can be employed to perform projection and exact-match selection. In terms of logic-based knowledge bases it can perform filtering of ground clauses. Selection on a conjunction of exact-match criteria is simply accomplished by incorporating all of them in the reference pattern. Disjunction-based selection could be done by using concatenated search patterns if the total length is less than n (and matching on a subset of the detectors), or by connecting more than one optical matcher in a pipeline.

Operations that access data repeatedly (like joins) and/or randomly (like sorting) cannot be implemented with a memory-less setup like the one described. Nevertheless, the global connectivity of optics can undoubtedly be exploited with other designs.

9.7.4. Future Work - Implementation of Relational Operations Using Optics

Future research in the area of optics will be focused on investigating various schemes for the efficient implementation of relational operations using optical devices such as spatial light modulator arrays, etalons and holographic memories.

The capabilities and limitations of the interconnect technology utilized in realizing a computational or signal processing unit play an important role in determining the speed and flexibility of the operations that can be achieved by that unit. Optical signals can flow through three-dimensional space to achieve the required interconnect pattern between elements of a two-dimensional data array before executing the desired operation between them. To examine more closely these advantages, three categories of operations must be considered.

The first category is that requiring single element operations like selection and comparison. In such computations, each element in a one- or two-dimensional array is processed independently from the rest of the array elements. The interconnectivity required by these operations is the loading and unloading of data to a processor array. Clearly, optical interconnections have the advantage of being able to input an entire data array in parallel using the third dimension of data propagation. On the other hand, in our electronic associative processor, data can be input and output only along the edges of a two-dimensional array, one row-column at a time. Optics have a lot to offer in D/KB systems where single-element operations are common.

Another category of operations is that of sorting, which is especially important in D/KB systems. Computations of this type require global interconnections between all the elements of the input array, that is, every element of the output array is dependent of all the entries in the input array. The structure of the sorting problem suggests an efficient algorithm in which computations grow as O(N*logN). In order to achieve these computational savings, complex interconnect configurations are necessary among the input elements of the array. Additionally, these interconnections have to be changed during the different stages of the computation. The requirement for dynamic interconnections can be exploited by employing the perfect shuffle function configuration. The perfect shuffle can be applied repeatedly at each stage of the computation to produce the currently desired interconnect pattern, presumably at the expense of extra time required to complete the interconnections. Optics offer the perfect shuffle function efficiently, hence its use in hardware sorting units would lead to improvements in system throughput.

The third category includes space and time variant operations like equi- and theta-join. The input relations can form two one-dimensional arrays and each element of the first array must be compared to all the elements of the second array. The interconnectivity pattern for these operations varies in space and time. Furthermore, the various interconnections are data dependent, making it impossible to predict in advance the appropriate interconnection patterns required at the different stages of the computation. The throughput of a parallel machine implementing this type of operations is critically affected by the availability of a dynamic and global interconnect network. Many processors could be idle for a significant number of cycles waiting for data to be properly routed to them. The overhead associated with the supervision of a controller in such a multiprocessor environment lacking space and time variant interconnection network may severely degrade all the advantages of parallel processing. Optics again offer great interconnection flexibility.

At a higher level, the use of electronic content addressable memory has been considered for improving the performance of database operations. Most of these efforts have not met with much success primarily because of the small size and the high cost of these devices and the slow data loading time. On the other hand, optical content addressable memories have the potential for holding megabytes of data at an appreciably lower cost. Since they are hologram-based their major disadvantage is that they are read-only. However, for very large data/knowledge bases indexing structures can be devised which are rather insensitive to updates provided that the update rate remains moderate. Thus, holographic content addressable memories could serve in the future for processing indices to very large databases. We are currently investigating this issue in a separate research effort. Finally, as the field develops, holographic memories may even be adopted as a primary storage medium.

9.8. An Architecture for Very Large Knowledge Bases

Existing Knowledge Bases (KB) have only a limited, relatively small size. In the near future, however, KB with data and rules on the order of 10^{11} - 10^{13} Bytes and an inference engine that can process hundreds of thousands of rules will be needed so that the next appropriate step is towards architectures for Very Large Knowledge Bases (VLKB). Obviously conventional techniques are not sufficient for the effective manipulation of such a vast amount of information and new powerful methods are required which will involve extensive parallel processing.

Currently, KBs are designed for specific problems such as bacterial infections or nuclear reactor control. As a result their application is limited. In contrast to these homogeneous, narrowly oriented systems, the future, general purpose VLKB will contain different types of information such as: multiple rule sets, many conventional and unconventional data bases, purely numerical data, formatted and unformatted text. This diversion calls for different types of processors too. For example, associative processors, data filters, relational operators, text processors, surrogate file processors etc. The incorporation of all these processing units in the same system along with the efficient integration of Logic and a Data Base Management System will be essential to any future design.

We are investigating various solutions for the management of very large data and knowledge bases in the support of multiple inferencing mechanisms for logic programming. The entire system must operate as a back-end machine removing from the host computer all the time-consuming operations for retrieving and manipulating data. As was previously pointed out, the evaluation of goals can require the accessing of the extensional database (EDB) of facts in very general ways and one must often resort to indexing on all fields of the facts. Cast in relational database terminology each relation must be indexed and each attribute of each relation must also be indexed.

The use of surrogate files helps to improve retrieval performance because less processing is required due to their smaller size. However, in some cases additional performance can be obtained by distributing the surrogate file entries as uniformly as possible over many disks to allow for parallel processing. We are developing a special Surrogate File Processor (SFP), that will utilize the query code word (QCW) as a search argument to obtain the list of unique identifiers that qualify.

A proposed architecture [BER87a] for this system involves several SFPs operating on the disks that contain the surrogate file. The unique identifiers are sent to an extensional data base manager which in turn retrieves the corresponding tuples from the disks containing the EDB.

9.8.1. The Concept of the Data/Knowledge Base Processor

Shown in Figure 9.8.1 is the block diagram of the system where the surrogate file and both the IDB and EDB are stored in the same group of disks which is controlled by a single Data Collector (DC). Processing is performed by the Data/Knowledge Base Processor (D/KBP) which is directly connected to the host computer.

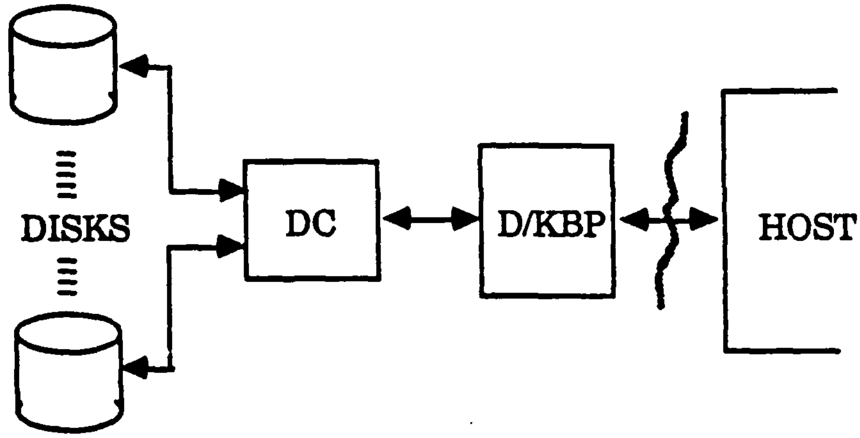


Figure 9.8.1 A Data/Knowledge Base Back-End System.

The D/KBP, the heart and the brains of the system, will be a specially designed piece of hardware that processes raw data coming from the disks performing various relational operations, data filtering, sorting, searching etc. It will encapsulate all the processing power necessary to manipulate the knowledge base including specialized hardware such as the SFP, sorting pipes, other relational operators and a general purpose processor. Its local memory will be large enough to accommodate the appropriate software for the inferencing mechanism and the data base management system as well as the qualified tuples that need further processing.

When the host issues a request for a transaction to the D/KBP, the data involved are located on the disks with the help of the SFP, retrieved and placed to the local memory by the general purpose processor. Then a combination of software and hardware techniques are employed in the D/KBP for the efficient data processing so that only useful information is returned to the host.

However, even this configuration is inadequate to handle hundreds of GBytes of data and unable to provide an acceptable I/O transfer rate. We envision a Very Large Knowledge Base Architecture (VLKBA) that will have about 500 GBytes of magnetic and 1500 GBytes of optical disk storage in its full configuration. The VLKBA is the topic of the next paragraph.

9.8.2. The Very Large Knowledge Base Architecture (VLKBA)

Shown in Figure 9.8.2 is an overall diagram of the initial design of a Very Large Knowledge Base Architecture. The VLKBA consists of a large number of secondary storage units (magnetic and optical disks, magnetic tapes) arranged in groups. Each group is controlled by a single data collector which receives data from multiple disks simultaneously and passes them to a Data/Knowledge Base Processor. All the D/KBP's have access to a large semiconductor memory which acts as a disk cache memory between the disks and the host machine.

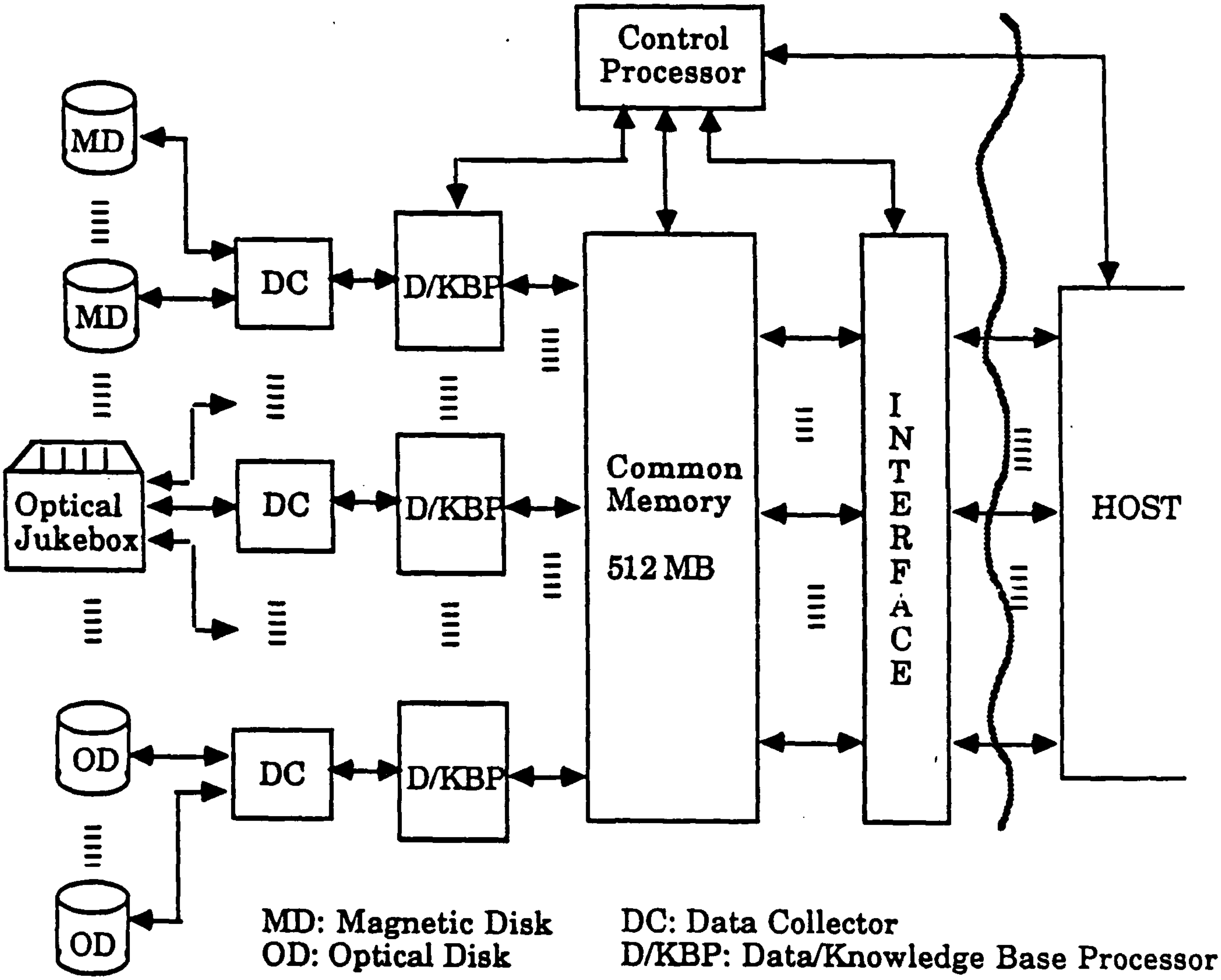


Figure 9.8.2 The Very Large Knowledge Base Architecture.

The D/KBPs communicate with each other and with the front-end computer through the common memory. The communication between the common memory and the host is established with an interface that allows for maximum bandwidth and arbitrary channel connection. The entire system is controlled by the Control Processor which accepts requests from the host and translates them to the appropriate commands for the VLKBA.

The primary goal of the design is to achieve maximum performance using a high degree of parallelism. Each part of the VLKBA is described separately in the following sections.

9.8.2.1. Memory Storage Units

In order to achieve such a huge capacity we can only consider the largest currently available mass storage media, that is, magnetic disks with movable heads and large optical disks. Some of the most important characteristics of these devices are shown in Table 9.8.1.

	MAGNETIC DISKS (Moving Heads)	OPTICAL DISKS (Large Diameter, Write-Once)
Capacity (GBytes)	1 - 5	5 - 10
Transfer Rate Burst (MB/sec)	3	0.7 - 10
Transfer Rate Sustained (MB/sec)	up to 3	0.2 - 1.0
Average Access Time (ms)	15 - 30	150 - 1000
Average Latency Time (ms)	8 - 10	20 - 60

Table 9.8.1. Secondary Memory Characteristics.

With the current capacity of the largest magnetic disks being on the order of 5 GBytes we need 100 such units to reach the desired 500 GBytes of magnetic storage.

In the optical disk area there is a greater variety. Optical disks provide significantly larger capacity but they have lower transfer rates. We are currently examining the possibilities for multiple-beam read from a single disk which could increase the I/O bandwidth dramatically. Another major disadvantage of the optical technology is the inability to change the information once it has been recorded on the optical surface but it seems that this problem will be overcome in the next few years.

The largest part (more than 1000 GBytes) of the optical storage will be provided by an optical "jukebox" [AMM85, ALT86]; a device that accommodates from 64 to 128 14-inch optical disks arranged in an on-line library configuration and accessed via an automated handling mechanism similar in concept to the well-known music jukebox. For the remaining 500 GBytes we are planning to use Write-Once Large (14") Optical Disks with a capacity of 10 GBytes/platter.

A group of disks may be interleaved to speed up data transfers in a manner analogous to the speedup achieved by main memory interleaving. Conventional disks may be used for interleaving by spreading data across disks and by treating multiple disks as if they were a single one. In the synchronized disk interleaving mode [KIM85,86], every page of the memory is distributed orthogonally over a group of M disks controlled by a common control unit. Every request for a specific page (or a block of more than one page) is broadcasted simultaneously to all M disks that execute the same transaction in parallel. The average service time (ST) for a request is given by:

$$ST = T_a + \frac{T_{tr}}{M}$$

where T_a is the average access time (average seek plus average latency time) and T_{tr} is the time to transfer (read or write) a block of data. Path delays due to rotational positioning sensing misses, which are significant in disk systems with skew distribution, are completely eliminated. Obviously, the performance of the design is improved when larger blocks of data are transferred. Synchronized disk interleaving provides simplified control because the interleaved disks can be manipulated as a single unit. Since the load is evenly balanced over all the devices, queuing delays due to multiple requests for a specific disk are avoided, thus allowing for maximum degree of parallelism and considerably lower service time. The reliability of the system can be also improved with minimum redundancy. A typical number for M is 10.

Each D/KBP will have access to 10 synchronized magnetic disks with an overall capacity of 50 GBytes. Every group of disks will be controlled by a separate Data Collector. The Data Collector will receive data from all the disks simultaneously thus obtaining a data transfer rate of about 30 MBytes/sec to each Data/Knowledge Base Processor. Thus, we will have 10 such groups and the total transfer rate can be as high as 300 MBytes/sec. Not shown in Figure 9.8.2 is the disk controller. We envision that each disk will have its own control processor and this processor will share the controller function with the Data Collector.

The average sustained transfer rate from the jukebox is a little below 50 MBytes/sec. Similarly, the low transfer rate from each of the other optical disk drives allows 16 such units to be serviced by a single data collector. Therefore, the output rate from the optical devices will be about 100 MB/sec raising the overall total for the Disks-To-D/KBPs bandwidth up to 400 MB/sec. However, as previously stated, we believe that the data rate from optical disks has the potential to be increased considerably through multi-beam reading. This speculation must await the results of further research.

Provision will be made for at least one magnetic tape unit for back-up purposes.

9.8.2.2. The Data/Knowledge Base Processors

The D/KBP accepts data from the data collector and either processes it or passes it through to the common memory. With regard to internal optimization the D/KBP must be able to generate and control index information for the data it manages, it must be able to optimally place the data on disk for minimization of access and update time and it must be able to maintain the data in terms of security, integrity, backup and recovery. Our work with the surrogate files, discussed in previous sections, will have a significant impact on the design of the D/KBP.

A more detailed block diagram of the D/KBP is shown in Figure 9.8.3.

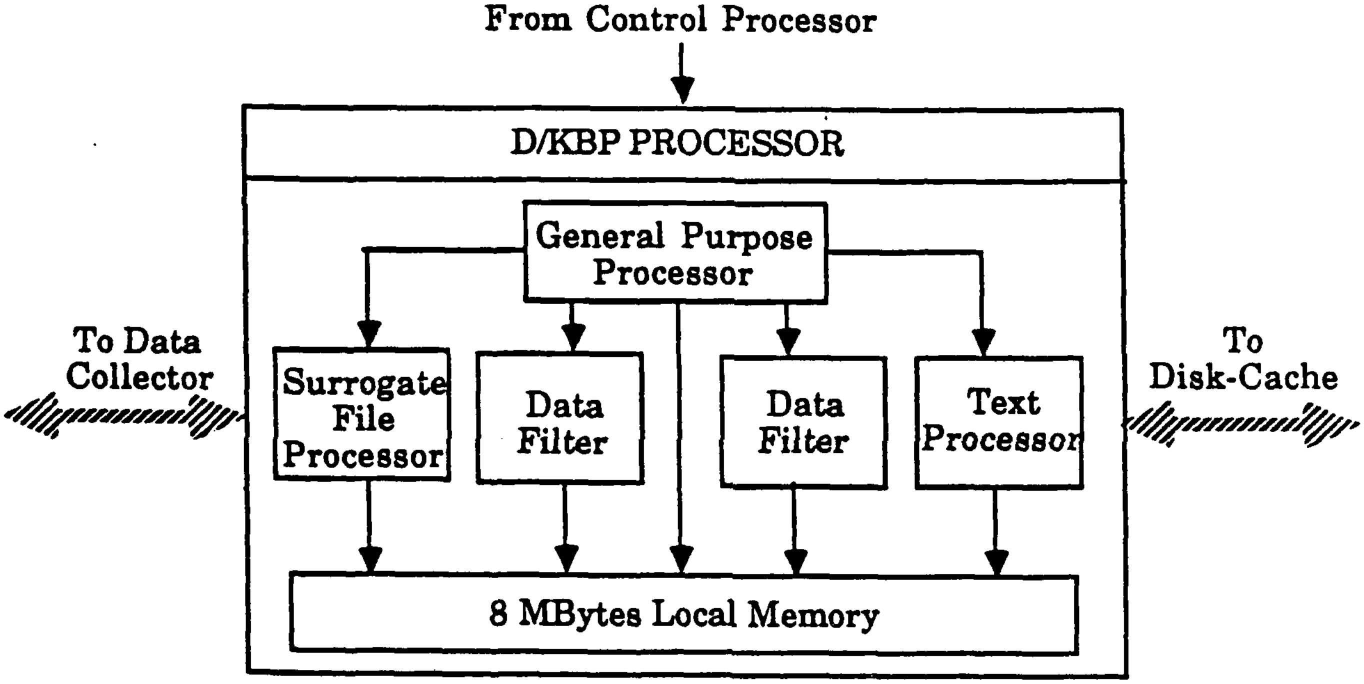


Figure 9.8.3 The Data/Knowledge Base Processor.

It will contain 8 MBytes of local memory and several specialized processors. The use of the Surrogate File Processor has already been illustrated. The General Purpose Processor will undertake a part of the internal control of the D/KBP and any other job that cannot be performed by the other processors (i.e. numerical computations). In addition the D/KBP contains a filter processor which performs the more common operations such as sort, merge, select, project and join as well as a special text processor.

The D/KBP performs two classes of operations on the data it controls. There are processes that respond to external commands from the control processor as shown in Figure 9.8.2 and internal processes that it must undertake to operate in a near optimal way. We believe that much of the inferencing capabilities of current AI systems will become part of the database system to form an intelligent database or expert database system or perhaps the term knowledge base system will take on that meaning in the future. We believe that new functions will be added to the database system to give it new functionality. For instance in work by Yokota and Itoh [YOK86] they discuss a relational knowledge base system that has the added functionality of unification-join and unification-restriction. The D/KBP will be designed to include appropriate addition capabilities.

There will be one D/KBP for each group of magnetic disks, three for the entire jukebox (because there are three different channels) and one for every group of optical disks bringing their total number to 16.

Returning to Figure 9.8.2 the nonprocessed or processed data are placed in the common memory. These data will be removed via the control processor for some applications but mostly via the interface to the host. The bandwidth between the D/KBP and the common memory and between the common memory and the interface will be on the order of 100's of MBytes/sec so as not to be a bottleneck.

9.8.2.3. The Common Memory

The use of a fast electronic buffer, as a disk cache, between the disks and the host offers many advantages; among them, higher bandwidth and synchronous communication. The size of this memory, which may be common to all the D/KBPs, lies in the order of 10^8 Bytes. It must accommodate multiple ports connected in parallel that read and write data simultaneously.

An initial design of such a system consists of a) the memory partitioned in M_b banks, where M_b is equal to the number of ports connected to it, and b) the appropriate interconnection network. Each page of the memory is orthogonally distributed over all the banks so that, a word with address (p,d) --where p is the page number and d the displacement in this page-- is located in the memory bank $M[d \textit{ mod } M_b]$ and its address in this bank is $(p*S_p + d) / M_b$, with S_p the size of a page in words. Every port i scans continuously the memory banks according to the sequence: i, i+1, ..., M_b-1, 0, 1, ..., i,... Such a distribution physically allows simultaneous access from all ports, even to the same page, without causing any conflicts among them nor any suspension. The access port speed should be equal to that of the host's main memory, or at least half as fast.

This multiport, multiple-access disk cache can significantly enhance the performance of the I/O system. Even if the overall Disk-to-D/KBP bandwidth is less than 400 MBytes/sec, the transfer rate from the interface to the front-end computer can be considerably higher especially when the disk-cache hit ratio approaches 1.

The interface should allow multiple (on the order of 100) ports from the host computer to be connected to the Common Memory. It will be a perfect shuffle interconnection network. The appropriate connections will be established according to signals from the Control Processor. Generally, more than one host might have access to the knowledge base simultaneously.

9.9. Applications and Research Issues

As pointed out earlier we are concerned with the management of very large data and knowledge bases in a multiple inferencing environment. However, the VLKBA will serve as a resource for many other interesting avenues of research. Among these are questions concerning the management of very large multimedia databases, the development of new embedded architectures, the comparative analysis of database indexing structures, the optimal reorganization of the database in response to usage and some of the more mundane questions concerned with concurrency control, back up, recovery and distribution. In addition, many problems have to be solved in order to achieve the desired cooperation between optical and magnetic equipment.

A promising field for future research is the all-optical data/knowledge base machine. The rapid advance of optical technology, especially in the optical interconnection networks, may soon lead to entirely new architectures for DBMS. Consider, for example, multiple laser beams reading on optical disks at data rates two orders of magnitudes faster than the current ones. This constant stream of data could be guided through optical fibers to an optical computer where many operations could be performed on the data prior to converting it to electronic pulses. Such a system would eliminate the need for data collectors, some of the large semiconductor memory and many of the processing units.

An additional use for VLKBA is the evaluation of experimental machines. When a machine is evaluated on the basis of performance (time per job, throughput, etc.) one must be able to keep the machine supplied with data. In fact, the data rate to the machine under test must be greater than the ability of the machine to handle it in order to obtain a realistic measure of performance. This requirement applies across the entire range of applications from processing intensive applications such as image processing to input/output intensive applications such as data base management. To obtain a realistic measure of a machine's performance, both processing and input/output time must be taken into consideration. Thus, if all of the data will not fit into the machine's main memory, the time to input new data must be taken into account in the performance measure. The time to complete the job then is the sum of the load and process time provided that they cannot be overlapped. In order to ensure a realistic test, the VLKBA must have sufficient capacity and bandwidth to supply the data for a large problem at stress rates to the machine under test.

For problems that are processing intensive, the VLKBA must be able to supply data to the machine being evaluated at the highest rate it can handle. Alternatively, suppose that the machine under test is input/output bound for problems of interest. Then, preprocessing can be performed in the VLKBA in order to enrich the data being sent to the machine under test. Thus, in addition to testing the machine, the VLKBA can identify some of the requirements of the secondary storage system to support the machine being evaluated.

The testing of machines places some severe constraints on the VLKBA. It must be able to sustain output data rates in the hundreds of MBytes per second range and have a data capacity in the hundreds of GBytes. It must have the facility to provide raw data to a machine under test at stress data rates and must also be able to perform considerable processing activities in order to enrich the data being sent to the machine under test. It must be able to interface with a wide variety of machines. It must have some level of reconfigurability so that the above functions can be performed, and it must be partitionable so that it can interact with more than one machine simultaneously.

9.10. References

[AMM85] G. J. Ammon, J. A. Calabria, D. T. Thomas, " A High-Speed, large-Capacity, "Jukebox" Optical Disk System," IEEE Computer, Vol. 18, No. 7, 1985, pp. 36-45.

[AHU80] S. R. Ahuja, C. S. Roberts, " An Associative/Parallel Processor for Partial Match Retrieval Using Superimposed Codes," Proc. Annual Symp. on Computer Architecture, 1980, pp. 218-227.

[ALT86] W. P. Altman, G. M. Claffie, M. L. Levene, " Optical Storage for High Performance Applications in the late 1980s and beyond," RCA Engineer, Vol. 31, Jan./Feb. 1986.

[BAU71] R. Bauer, E. McCreight, " Organization and Maintenance of Large Ordered Indexes," Acta Informatica, Vol. 1, 1972, pp. 173-189.

[BER87a] P. B. Berra, S. M. Chung, N. I. Hachem, " Computer Architecture for a Surrogate File to a Very Large Data/Knowledge Base," IEEE Computer Vol. 20, No.3, 1987, pp. 25-32.

[BER87b] P. B. Berra, N. B. Troullinos, " Optical Techniques and Data/Knowledge Base Machines," IEEE Computer Vol. 20, No. 10, 1987, pp. 59-70.

[BIT83] D. Bitten, H. Boral, et al., " Parallel Algorithms for the Execution of Relational Database Machine Operations," ACM Trans. Database Systems, Vol. 8, No. 3, 1983, pp. 324-353.

[BRA84] K. Bratbergsengen, " Hashing Methods and Relational Algebra Operations," Proc. VLDB, 1984, pp. 323-333.

[CAR75] A. F. Cardenas, " Analysis and performance of Inverted Database Structures," CACM, Vol. 18, No. 5, 1975, pp. 253-263.

[CAR79] J. L. Carter, M. N. Wegman, " Universal Class of Hash Functions," Journal of Computer and System Sciences, Vol. 18, No. 2, 1979, pp. 143-154.

[CAR84] D. B. Carlin, J. P. Bednarz, et al., " Multichannel Optical Recording Using Monolithic Arrays of Diode Lasers," Applied Optics, Vol. 23, No. 22, 1984, pp. 3994-4000.

[CHU87] S. M. Chung, P. B. Berra, " Surrogate File Structures for Very Large Data/Knowledge Bases," submitted to a conference, 1987.

[CLA86] K. Clark and S. Gregory, " PARLOG: Parallel Programming in Logic, "ACM Trans. on Programming Languages and Systems, Vol. 8, No. 1, 1986, pp. 1-49.

[CON87] J. S. Conery, Parallel Execution of Logic programs, Kluwer Academic Publishers, Boston, 1987.

[COL86] R. M. Colomb, " A Hardware-Intended Implementation of Prolog Featuring a General Solution to the Clause Indexing Problem," Ph. D. Dissertation, University of New South Wales, October 1986

[DAT86] C. J. Date, An Introduction to Database Systems, Volume 1, Chap. 21, Addison-Wesley Systems Programming Series, 1986.

[DIG82] Digital Equipment Corporation, RA 81 Disk Drive User Guide, 1982

[DWO84] C. Dwork, P. Kanellakis, J. Mitchell, " On the Sequential Nature of Unification," Journal of Logic Programming, Vol. 1, 1984, pp. 35-50.

[FAG79] R. Fagin, J. Nievergelt, et al.," Extendible Hashing - A Fast Method for Dynamic Files," ACM Trans. Database Systems, Vol. 4, No. 3, 1979, pp. 315-344.

[FAL84] C. Faloutsos, S. Christodoulakis, " Signature Files: An Access Method for Documents and Its Analytical Performance Evaluation," ACM Trans. on Office Information Systems, Vol. 2, No. 4, 1984, pp. 267-288.

[FAL85] C. Faloutsos, S. Christodoulakis, " Design of a Signature File Method That accounts for Non-Uniform Occurrence and Query Frequencies," Proc. VLDB, 1985, pp.165-170.

[GAR86] A. K. Garg, C. C. Gotlieb , " Order-Preserving Key Transformations," ACM Trans. on Database Systems, Vol. 11, No. 2, 1986, pp 213-234.

[GAY85] T. K. Gaylord, M. M. Mirsalehie, C. C. Guest, " Optical Digital Truth-table Look-up Processing," Optical Engineering, Vol. 24, Jan./Feb., 1985, pp. 48-58.

[GUI86] P. S. Guifoyle, W. J. Wiley, " Combinatorial Logic Based Optical Computing," Proc. SPIE, Vol. 639-17, April, 1986.

[HAC88] N. I. Hachem, P. B. Berra, " Back End Architecture based on Transformed Inverted Lists, A Surrogate File Structure for a Very Large Data/Knowledge Base," HICSS21, to appear January 1988.

[KIM85] M. Kim, " Parallel Operation of Magnetic Disk Storage Devices: Synchronized Disk Interleaving," Proc. Int'l Workshop on Database Machines, 1985, pp. 300-330.

[KIM86] M. Kim, " Synchronized Disk Interleaving," IEEE Trans. on Computers, Vol. C-35, No. 11, 1986, pp. 978-988.

[KIT83] M. Kitsuregawa, H. Tanaka, T. Moto-Oka, " Application of Hash to Data Base Machine and Its Architecture," New Generation Computing, Vol. 1, 1983, pp. 63-74.

[LAR79] P. -A. Larson, " Analysis of Repeated Hashing," BIT 20, 1980, pp.25-32.

[LAR80] P. -A. Larson, " Linear Hashing with Partial Expansions," Proc. VLDB, 1890, pp. 224-232.

[LAR81] P. -A. Larson, " Analysis of Index-Sequential Files with Overflow Chaining," ACM Trans. on Database Systems, Vol. 6, No. 4, 1981, pp 671-680.

[LAR82] P. -A. Larson, " Performance Analysis of Linear Hashing with Partial Expansions," ACM Trans. on Database Systems, Vol. 7, No. 4, 1982, pp. 566-587.

[LIT80] W. Litwin, " Linear Hashing: A New Tool for File and Table Addressing," Proc. VLDB, 1980, pp. 212-223.

[LEE86] D. L. Lee, " A Word-Parallel, Bit-Serial Signature Processor for Superimposed Coding," Proc. Int'l Conf. on Data Engineering, 1986, pp. 352-359.

[LLO84] J. W. Lloyd, Foundations of Logic Programming, Springer-Verlag, 1984.

[MAL85] J. Maluszynski, H. Jan Komorowski, " Unification-free Execution of Logic Programs," Proc. of Int'l. Conf. on Logic Programming, 1985.

[MAR77] J. Martin, Computer Data-Base Organization, second edition, Prentice -Hall, Inc, NJ, 1977, Chapter 20.

[MAR82] A. Martelli, U. Montanari, " An Efficient Unification Algorithm," ACM Trans. on Programming Languages and Systems, Vol. 4, No. 2,. 1982, pp. 258-282.

[MOR86] Y. Morita, H. Yokota, H. Itoh, " Retrieval-By-Unification Operation on a Relational Knowledge Base," Proc. of 12th VLDB, 1986, pp. 52-59.

[MOS87] Y. S. Abu-Mostafa, D. Psaltis, " Optical Neural Computers," Scientific American, March, 1987, pp. 88-95.

[OUK83] M. Ouksel, P. Scheuermann, " Storage Mappings for Multidimensional Linear Dynamic Hashing," Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, 1983, pp. 90-105.

[PFA80] J. L. Pfaltz, W. J. Berman, E. M. Cagley, " Partial-Match Retrieval Using Indexed Descriptor Files," CACM, Vol. 23, No. 9, 1980, pp. 522-528.

[PAT78] M. S. Paterson, M. N. Wegman, " Linear Unification," Journal of Computer and System Sciences 16, 1978, pp. 158-167.

[RAM84] K. Ramamohanarao, R. Sacks-Davis, " Recursive Linear Hashing," ACM Trans. on Database Systems, Vol. 9, No. 3, 1984, pp. 369-391.

[RAM86] K. Ramamohanarao, J. Shepherd, " A Superimposed Codeword Indexing Scheme for Very Large Prolog Databases," Proc. Int'l Logic Programming Conference, 1986, pp. 569-576.

[ROB65] J. A. Robinson, " A Machine-Oriented Logic Based on the Resolution Principle," Journal of the ACM, Vol.12, 1965, pp. 23-44.

[ROB79] C.S. Roberts, " Partial Match Retrieval via the Method of Superimposed Codes," Proceedings of the IEEE, Vol. 67, No. 12, 1979, pp. 1624-1642.

[SAC83] R. Sacks-Davis, K. Ramamohanarao, " A Two Level Superimposed Coding Scheme for Partial Match Retrieval," Information Systems Vol. 8, No. 4, 1983, pp. 273-280.

[SHI87] D. Shin, P.B. Berra, " An Architecture for Very Large Rule Bases Based on Surrogate Files," Proc. Int'l Workshop on Database Machines, 1987, pp. 555-568.

[SHO86] Y. Shobatake, H. Aiso, " A Unification Processor Based on a Uniformly Structured Cellular Hardware," Proc. of Int'l. Symp. on Computer Architecture, 1986, pp. 140-148.

[STO86] C. D. Stormon, " An Associative Processor and Its Application to Logic Programming Computation," CASE Center TR 8611, Syracuse University, 1986.

[VIT86] J. S. Vitter, R. A. Simons, " New Classes for Parallel Complexity: A Study of Unification and Other Complete Problems for P ," IEEE Trans. on Computers, Vol. C-35, No. 5, 1986, pp. 403 -418.

[WAD87] M. Wada, Y. Morita, et al., " A Superimposed Code Scheme for Deductive Databases," Proc. Int'l Workshop on Database Machines, 1987, pp. 569-582.

[WAR83] D. H. D. Warren, "An Abstract Prolog Instruction Set," Technical Note 306, SRI International, October 1983.

[WIS84] M. J. Wise and D. M. W. Powers, " Indexing Prolog Clauses via Superimposed Code Words and Field Encoded Words," Proc. International Symposium on Logic Programming, 1984, pp.203-210.

[WOO85] N. S. Woo, "The Architecture of the Hardware Unification Unit and An Implementation," Micro 18 Proc., 1985, pp. 89-98.

[YAO77] S. B. Yao, " Approximating Block Access in Database Organization," CACM, Vol. 20, No. 4, 1977, pp. 260-261.

[YOK86] H. Yokota, H. Itoh, " A Model and an Architecture for a Relational Knowledge Base," Proc. Annual International Symposium on Computer Architecture, 1986, pp. 2-9.

Appendix 9-A

Surrogate File Structures
for
Very Large Data/Knowledge Bases

Soon Myoung Chung
P. Bruce Berra

Dep't. of Electrical and Computer Engineering
Syracuse University
Syracuse, NY 13244-1240
U. S. A.

(315) 423-4445
berra@sutcase.case.syr.edu
chung@sutcase.case.syr.edu

This work was supported by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under Contract No. F30602-85-C-0008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC).

ABSTRACT

Surrogate files are very useful as an index for very large knowledge bases to support multiple logic programming inference mechanisms because of their small size and simple maintenance requirement. In this paper, we analyse the superimposed code word (SCW) and concatenated code word (CCW) surrogate file techniques in terms of storage space and time to answer queries in various cases. One of the most important results of our analysis is that the size and the query response time of the CCW is smaller than those of the SCW when the average number of arguments specified in a query is small. It is also shown that most of the query response time is used for the surrogate file processing when the extensional database is very large. Therefore, if we use a special architecture to speed up the surrogate file processing, the total query response time can be reduced considerably.

1. Introduction

Knowledge based systems consist of rules, facts and an inference mechanism that can be utilized to respond to queries posed by users. The objective of such systems is to capture the knowledge of experts in particular fields and make it generally available to nonexpert users. The current state of the art of such systems is that they focus on narrow domains, have small knowledge bases and are thus limited in their application.

As these systems grow, increased demands will be placed on the management of their knowledge bases. The intensional database (IDB) of rules will become large and present a formidable management task in itself. But, the major management activity will be in the access, update and control of the extensional database (EDB) of facts because the EDB is likely to be much larger than the IDB. The volume of facts is expected to be in the gigabyte range, and we can expect to have general EDB's that serve multiple inference mechanisms. In this paper we assume that the IDB is a set of rules expressed as logic programming clauses and the EDB is a relational database of facts.

In order to set the stage for the problem that we are interested in, consider the following simple logic programming problem:

1. grandfather(X,Y) ← father(X,Z), parent(Z,Y)
2. parent(X,Y) ← father(X,Y)
3. parent(X,Y) ← mother(X,Y)
4. father(pat, tiffany) ←

   father(don, louise) ←

   .

   .

.

5. mother(mary, louise) ←

mother(lisa, tiffany) ←

.

.

.

6. ← grandfather(X, joan)

The first three clauses form the IDB of rules for this problem, the next two sets form the EDB of facts and the last statement is the goal. To solve the problem (satisfy the goal), we must find the names of the grandfathers of joan. For this we search the father and mother facts on the second argument position, finding values for the first argument position that can be used later. Thus, we need to find joan's mother and father before finding her grandfathers. If we ask a similar but slightly different query

← grandfather(tom, X)

we search the first argument of the father and mother facts in attempting to satisfy it.

Consider the following general goal statement of a logic programming language

$$\leftarrow r\ (X_1, X_2, \cdots, X_n).$$

In this case, values for some subset of the X_i's will be given in the process of trying to satisfy its goal. Since the subset of the X_i's is not known in advance and can range from one to all of the values, this places considerable requirements on the relational database management system that supports the logic programming language. In fact, in order to insure minimum retrieval time from the relational database all of the X_i's must be indexed. With general indexing the index data could be as large as the actual EDB. In order to considerably reduce the amount of index data yet provide the same capability, we have considered surrogate files. Obviously if not all of the X_i's can take part in goal satisfaction then the indexing strategy will change, however in this paper we will assume the most general case in which all of the X_i's are active.

Retrieving the desired rules and facts in this context is an extension of the multiple-key attribute partial match retrieval problem because any subset of argument positions can be specified in a query and matching between terms consisting of variables and functions as well as constants should be tested as a preunification step.

In the context of very large knowledge bases the question arises as to how to obtain the desired rules and facts in the minimum amount of time. Two reasonable choices of indexing schemes to speed up the retrieval are superimposed code word (SCW) and concatenated code word (CCW) surrogate file techniques discussed by D. Shin et al. [21] and P.B. Berra et al. [2]. Surrogate files are constructed by transformed binary codes where the transform is performed by well chosen hashing functions on the original terms. In [2], SCW, CCW and transformed inverted list (TIL) surrogate files were discussed in terms of the structures, updating procedures, performance of relational operations on the surrogate files, and possible architectures to support them. The term "surrogate file" dates back to early work in information retrieval and other terms, such as "signature file" and "descriptor file" have been used to describe similar structures. [7, 8, 22]

Compared with other full indexing schemes such as inverted lists [4], SCW and CCW surrogate file techniques yield much smaller amounts of index data; about 20% of the size of the EDB [2] while the inverted lists may be as large as the EDB. This size advantage can yield increased retrieval performance especially when the number of search arguments is greater than one. Inverted lists show advantage in retrieval when a single argument is given since only one list need be processed. Surrogate file technique based on SCW or CCW can be easily implemented with parallel computer architectures because their structures are quite regular and compact. [1, 2, 14]

In terms of maintenance the surrogate file shows considerable advantages. When a new tuple is added to a relation the SCW or CCW is generated and added to the surrogate file. In the case of inverted lists each list must must be processed. Similar operations must be performed for deleting tuples from a relation. When changes to an existing tuple are made, the surrogate file entry must be changed and the proper inverted lists must be changed.

An important advantage of SCW and CCW surrogate file techniques is that they can be easily extended for the indexing of the rules expressed as Prolog clauses, where the matching between constants, variables, and structured terms is required to test the unifiability. M. J. Wise et al.[25], K. Ramamohanarao et al. [17], and M. Wada et al. [24] extended the SCW structure for the indexing of Prolog clauses and D. Shin et al. [21] extended the CCW structure to index the rules and facts in unified manner.

In this paper, we analyse SCW and CCW surrogate file techniques on the basis of storage space required for the surrogate file and time to retrieve the desired facts from the EDB. We limit our discussion to the EDB because most of the query response time is spent for fact retrieval and relational operations on the EDB, and the proposed structures of SCW and CCW for rule indexing are quite different, so it is difficult to make meaningful comparison at this time.

In the next section we introduce the structures and retrieval procedure of the SCW and CCW surrogate file techniques. We then develop the equations for the surrogate file size and the query response time. The analyses based on the simulation results are discussed next and finally we close with some thought on the performance improvement with SCW and CCW surrogate file techniques using a special architecture.

2. System Model for SCW and CCW Techniques

2.1. Superimposed Code Word (SCW)

Let a fact D contain A_r argument values, $D=\{d_1,d_2, \cdots ,d_{A_r}\}$. Each argument value ($d_i$, $1 \leq i \leq A_r$) can be mapped into a binary representation by a well chosen hashing function. The binary representation can be converted to another binary representation with pre-defined length and pre-defined weight, called a binary code word (BCW), by using a pseudo random number generator. The weight of a binary representation is the number of 1's in the binary representation. The process of generating a BCW from an argument value is well described in [18] by C. S. Roberts. The SCW of a fact is generated by ORing A_r BCW's obtained from A_r argument values. A unique identifier is then attached to the SCW and the fact. This unique identifier serves as a link between the two and is used as a pointer to the EDB or can be converted to a actual pointer to the EDB by dynamic hashing schemes such as linear hashing. [11, 12, 13]

Suppose we have a fact type called borders which is given as follows:

borders (Country_1, Country_2, Body_of_Water).

For a particular instance

borders (korea, china, yellow sea).

We would first hash the individual values,

H(korea)   H(china)   H(yellow sea)
↓   ↓   ↓
100...01   010...00   110...00

then the SCW would be formed as follows:

H(korea) → 100...01
H(china) → 010...00
H(yellow sea) → 110...00
110...01|00...01

with the binary representations logically ORed together. The unique identifier is attached as shown and the vertical line shows the boundary.

The retrieval process with the SCW surrogate file technique is as follows:

1) Given a query, obtain a query code word (QCW) by ORing BCW's corresponding to argument values specified in the query.

2) Obtain a list of unique identifiers to all facts whose SCW's satisfy

$$QCW{=}QCW \text{ .AND. } SCW$$

that is, obtain a list of all SCW's that have 1's in the same position as the QCW by sequentially ANDing the QCW with all entries in the SCW file.

3) Retrieve all the facts pointed to by the unique identifiers obtained in step 2 and discard the facts not satisfying the query. These are called "false drops". The facts satisfying the query are called "good drops". The false drops are caused by the non-ideal property of hashing functions and the logical ORing of BCW's which make facts with different argument values have the same SCW.

4) Return the good drops to the user.

2.2. Concatenated Code Word (CCW)

The CCW of a fact is generated by simply concatenating the binary representations (BR's) of all argument values and attaching the unique identifier of the fact. With the same example used for SCW, the CCW would be formed as

100...01|010...00|110...00|00...01.

The retrieval process with the CCW surrogate file is as follows:

1) Given a query, obtain a query code word (QCW) by concatenating BR's corresponding to argument values specified in the query. The portion of the

query code word for argument values which is not specified in the query is filled with don't care symbols.

2) Obtain a list of unique identifiers to all facts whose CCW's satisfies

$$QCW=CCW$$

by sequentially comparing the QCW with all CCW's in the CCW file. Note in this case the matching is performed on both 1's and 0's.

3) Retrieve all facts pointed to by the unique identifiers obtained in step 2 and compare the corresponding argument values of the facts with the query values to discard the false drops caused by the non-ideal property of hashing functions.

4) Return the good drops to the user.

3. Storage Requirement and Retrieval Performance of SCW and CCW Techniques

Storage requirements can be expressed by the size of surrogate files and retrieval performance can be measured by the query response time for a given query. Notations that are frequently used in this paper are shown in Table 3.1.

Notations	Meanings
A_r	Number of arguments in a fact
R_q	Average number of arguments specified in a query
GD	Average number of good drops per query
FD	Average number of false drops per query
S_{db}	Size of the extensional database in bytes
NR	Number of facts in the extensional database
NSB	Number of blocks in surrogate files
NDB	Average number of extensional database blocks retrieved
S	Size of surrogate file in bits
B	Size of a block in bytes
BR	Binary representation
BCW	Binary code word
b_{bcw}	Bit length of a binary code word
QT	Query Response time
T_{sp}	Surrogate file processing time
T_{dp}	Extensional database processing time
T_{ba}	A block access time
C_i	Value distribution factor, that is, the average number of facts which have the same value in the i-th argument
C_g	Average of value distribution factor (Average redundancy)

Table 3.1. Summary of Notations Frequently Used

3.1. Size of SCW and CCW Surrogate Files

The equations for the size of the SCW and CCW files are obtained in this section under the assumption, that, if the input values are different from each other, the selected hashing function maps those values into different output values, that is, there are no collisions by the hashing function.

With the above assumption, C. S. Roberts [18] presented the optimal bit length of a BCW in a SCW in terms of the number of arguments in a fact (A_r), the average number of arguments specified in a query (R_q), the number of facts in the EDB (NR), and the average number of false drops (FD). The equation for the bit length of a BCW (b_{bcw}) is given as

$$b_{bcw} = \left\lceil \frac{A_r}{R_q} \frac{[\ln(NR) - \ln(FD)]}{[\ln(2)]^2} \right\rceil . \tag{3.1}$$

The SCW also contains its unique identifier which must be greater than or equal to $\log_2 N$, thus the minimal bit length of a SCW (b_{scw}) is

$$b_{scw} = (b_{bcw} + \lceil \log_2 NR \rceil) . \tag{3.2}$$

Hence, the minimal size of the SCW file (S_{scw}) is as follows:

$$S_{scw} = b_{scw} \times NR. \tag{3.3}$$

For a CCW file, the minimal size can be derived from the fact that the bit length of the hashing function output for an argument in a fact must be at least $\left\lceil \log_2 \frac{NR}{C_i} \right\rceil$ where C_i, called the value distribution factor, is the average number of facts whose i-th arguments have the same value. From this fact, we can derive the minimal sizes of the CCW file (S_{ccw}).

A CCW contains the binary representation of each argument value and its unique identifier. Hence, the minimal size of the CCW file is

$$S_{ccw} = (\sum_{i=1}^{A_r} \left\lceil \log_2 \frac{NR}{C_i} \right\rceil + \left\lceil \log_2 NR \right\rceil) \times NR. \tag{3.4}$$

In this paper, we assumed that the hashing function is ideal and simulated the minimal storage requirement for surrogate files. However, in actual cases, the hashing function is not ideal and there will be around 30% of collisions if the number of the distinct hashing function output is equal to the number of distinct input argument values.[15] As we increase the length of the binary representation of an argument value, the probability of collisions will decrease: for example, if we assign two more bits to the binary representations in CCW surrogate file, the probability of the collision will be less than 10% and the net increase in surrogate file size will be around 1.5% of the EDB size.

3.2. Query Response Time Using SCW and CCW Surrogate Files

The query response time depends largely on the size of surrogate files and the method of obtaining pointers from the surrogate files. For the size of surrogate files, the equations derived in the previous section are used. Also, it is assumed that a sequential uniprocessor is available for surrogate file processing.

In general, the retrieval process using surrogate files can be divided into several sub-processes as follows:

1. Access to the surrogate files to read the code words from those files.
2. Comparing the QCW of a query with code words and obtaining a list

of pointers (unique id's) to the EDB.

3. Access to the EDB to read the facts pointed to by the pointers obtained in 2.

4. Comparing the query with the facts retrieved from the EDB. This step is to discard the false drops.

3.2.1. Surrogate File Processing Time

Let B be the size of a block in bytes, then there are

$$\mathrm{NSB} = \left\lceil \frac{S_{scw} \text{ (or } S_{ccw})}{8 \times B} \right\rceil \qquad (3.5)$$

blocks in surrogate files and each block contains $\left\lfloor \frac{\mathrm{NR}}{\mathrm{NSB}} \right\rfloor$ code words except the final block. Initially, the first block of the surrogate file is accessed in

$$\begin{aligned} T_{ba} = {} & \text{Average seek time} + \text{Rotational delay} \\ & + \frac{B}{\text{Transfer rate}} \end{aligned} \qquad (3.6)$$

and the first block will be searched in

$$\begin{aligned} T_{ss} = {} & \text{the number of bytes in a QCW} \\ & \times \text{time for byte comparison} \;/\; 2 \times \left\lfloor \frac{\mathrm{NR}}{\mathrm{NSB}} \right\rfloor \\ & + \frac{(\mathrm{GD}+\mathrm{FD})}{\mathrm{NSB}} \times (\text{uid collection time}) \end{aligned} \qquad (3.7)$$

where GD denotes the average number of good drops and FD denotes the average number of false drops per query.

If we assume that the blocks in the surrogate file reside consecutively in a disk, then the time for accessing the remaining (NSB − 1) blocks is

$$\begin{aligned} T_{sa} = {} & \frac{B}{\text{Transfer rate}} \times (\text{NSB} - 1) \\ & + \text{Rotational delay} \times (\text{\# of tracks for SF} - 1) \\ & + \text{Minimum seek time} \times (\text{\# f cylinders for SF} - 1)\,. \end{aligned} \tag{3.8}$$

If sufficient number of buffers are provided, the reading of the last (NSB −1) blocks can be overlapped with the searching of the first (NSB − 1) blocks. Therefore, the maximum of these two times, i.e, max $(T_{sa}\,,\, T_{ss} \times (\text{NSB} - 1))$ will contribute to the surrogate file processing time. For the last block of the surrogate file, the searching time is not overlapped with the block access time. Thus, the total surrogate file processing time is

$$T_{sp} = T_{ba} + \max\,(T_{sa}\,,\, T_{ss} \times (\text{NSB} - 1)) + T_{ss}\,. \tag{3.9}$$

Here we ignore the buffer switching time and the process wake-up time, i.e, the overhead time caused by buffering.

3.2.2. Extensional Database Processing Time

The average number of accesses to the EDB per query is the summation of the average number of good drops and the average number of false drops. If the facts to be retrieved are assumed to be randomly distributed over the EDB, the average number of EDB blocks to be retrieved is

$$\text{NDB} = \left\lceil \frac{S_{db}}{B} \right\rceil \times \left(1 - \left(1 - \frac{1}{\left\lceil \frac{S_{db}}{B} \right\rceil}\right)^{GD+FD}\right). \tag{3.10}$$

where S_{db} denotes the size of the EDB in bytes. If we assume that attributes are independent within a relation, GD can be approximated by using C_i and NR.

$$GD = \begin{cases} NR \prod_{i \in R_q} (\frac{C_i}{NR}) & \text{if } GD > 1 \\ 1 & \text{otherwise} \end{cases} \tag{3.11}$$

Once a EDB block is retrieved, then the facts with matched unique id's will be compared with the query to discard the false drops. The time for this comparison is

$$T_{dc} = \frac{(GD + FD)}{NDB} \times \text{the number of bytes in a fact} \tag{3.12}$$
$$\times \text{ time for byte comparison } / \ 2 \ .$$

EDB block accessing and comparison can also be overlapped. So if we assume that the EDB blocks are randomly accessed, the total EDB processing time is

$$T_{dp} = T_{ba} + \max(T_{ba}, T_{dc}) \times (NDB - 1) + T_{dc} \ . \tag{3.13}$$

However, since the EDB blocks are randomly accessed, the block access time is much more than the block comparison time. Therefore, the total EDB processing time can be simplified as

$$T_{dp} \approx T_{ba} \times NDB \ . \tag{3.14}$$

3.2.3. Query Response Time

The query response time for a given query is the summation of all the surrogate file processing time and the EDB processing time.

$$QT_{scw} \text{ (or } QT_{ccw}) = T_{sp} + T_{dp} \qquad (3.15)$$

4. Simulation and Analysis for SCW and CCW Techniques

Simulations are performed with the equations for the size of surrogate files and the query response time using SCW and CCW techniques assuming that the surrogate files are consecutively stored in a disk, the EDB are randomly stored in a number of disks and the block load factor of the surrogate file and the EDB is 1. If the EDB is dynamic, then the block load factor will be lowered and consequently the number of blocks to be accessed will increase somehow. But once a block is accessed, the time for processing a block will decrease.

4.1. Surrogate File Size

For the simulation of the surrogate file size , it is assumed that the EDB remains at the same size regardless of variation of A_r and 15 bytes are used for each argument value. Therefore, NR, the number of facts in the EDB, can be calculated as follows:

$$NR = \left\lfloor \frac{S_{db}}{15 \times A_r} \right\rfloor$$

where S_{db} represents the actual EDB size not including the unique identifiers for each fact of the EDB. We also assumed that each argument of a fact in the EDB has the same redundancy value, C_g, which is the average of the C_i's:

$$C_g = \frac{\sum_{i \in A_r} C_i}{A_r}.$$

The results for the size simulation are shown in Figures 4.1 through 4.3. In Figure 4.1 we plot the size of the SCW surrogate file (S_{scw}) as a function of the number of arguments (A_r) in a fact. The size of the surrogate file is expressed as a percentage of the EDB. The EDB sizes are 10^5, 10^7 , and 10^9 bytes while the average number of arguments in a query (R_q) takes on the values one and two. Note that S_{scw} increases with the size of the EDB (S_{db}) but decreases with R_q. The reasons for this behavior are readily apparent from equations 3.1 to 3.3.

In SCW case, if we allow more false drops then the length of the SCW becomes shorter which results in a smaller S_{scw}. However, more false drops leads to more EDB accesses.

In designing the SCW surrogate file one must set the expected number of arguments in a query. In terms of size, the worst case of course is when R_q is 1 and as the value for R_q is set at progressively higher values S_{scw} becomes very small. However, if we assume large R_q in designing the SCW file, we have to allow more false drops than the expected number of false drops, FD, whenever the number of arguments specified in a query is smaller than R_q. [18]

In Figure 4.2 we plot the size of the CCW surrogate file(S_{ccw}) as a function of the average redundancy(C_g) in the data. Note that with greater redundancy S_{ccw} becomes smaller because a smaller number of bits can be used for each binary code word. Also note that S_{db} and A_r have significant effects on S_{ccw}.

Finally, in Figure 4.3 we compare S_{scw} and S_{ccw} for various conditions. With regard to the size of surrogate files, we can say that the CCW file technique is better than the SCW technique, even though S_{scw} .nay be smaller than S_{ccw} when R_q is large, because we assumed that the average number of arguments specified in a query is usually not more than 2. However, in both cases the surrogate file is generally less than 20% of the size of the EDB.

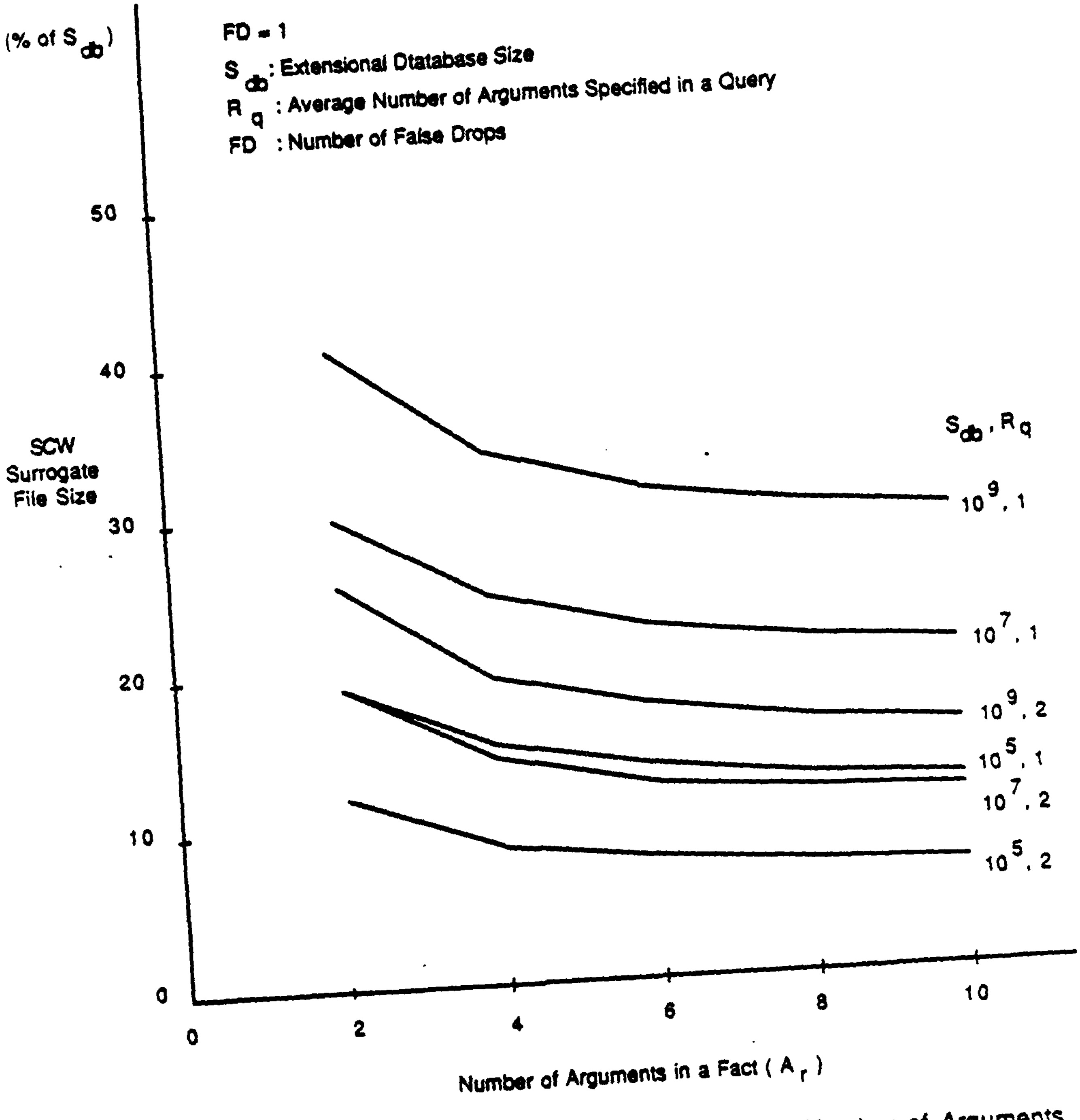


Figure 4.1 Effect of EDB Size and the Average Number of Arguments in a Query on the SCW Surrogate File Size

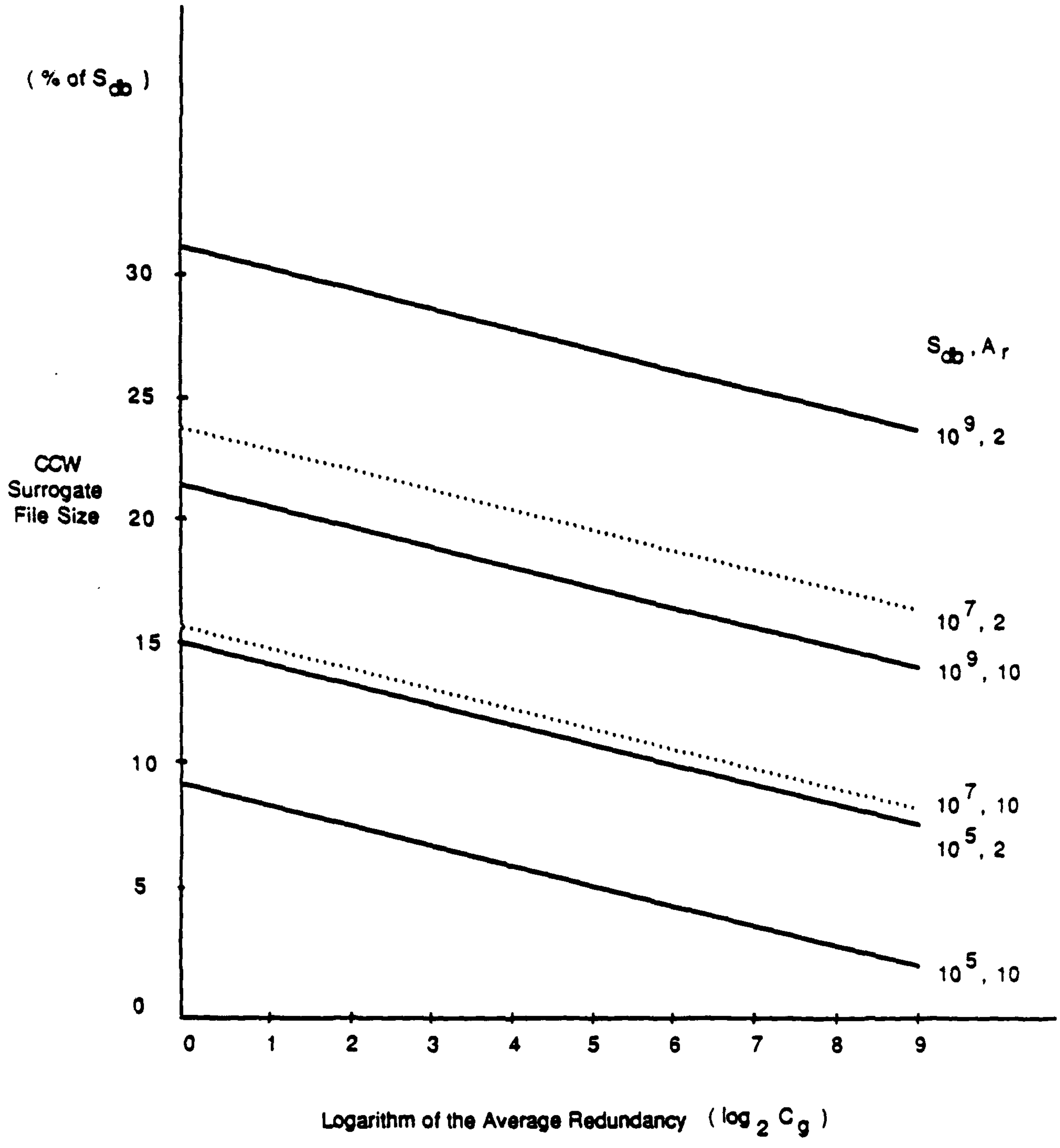


Figure 4.2 Effect of the Average Redundancy on the CCW Surrogate File Size

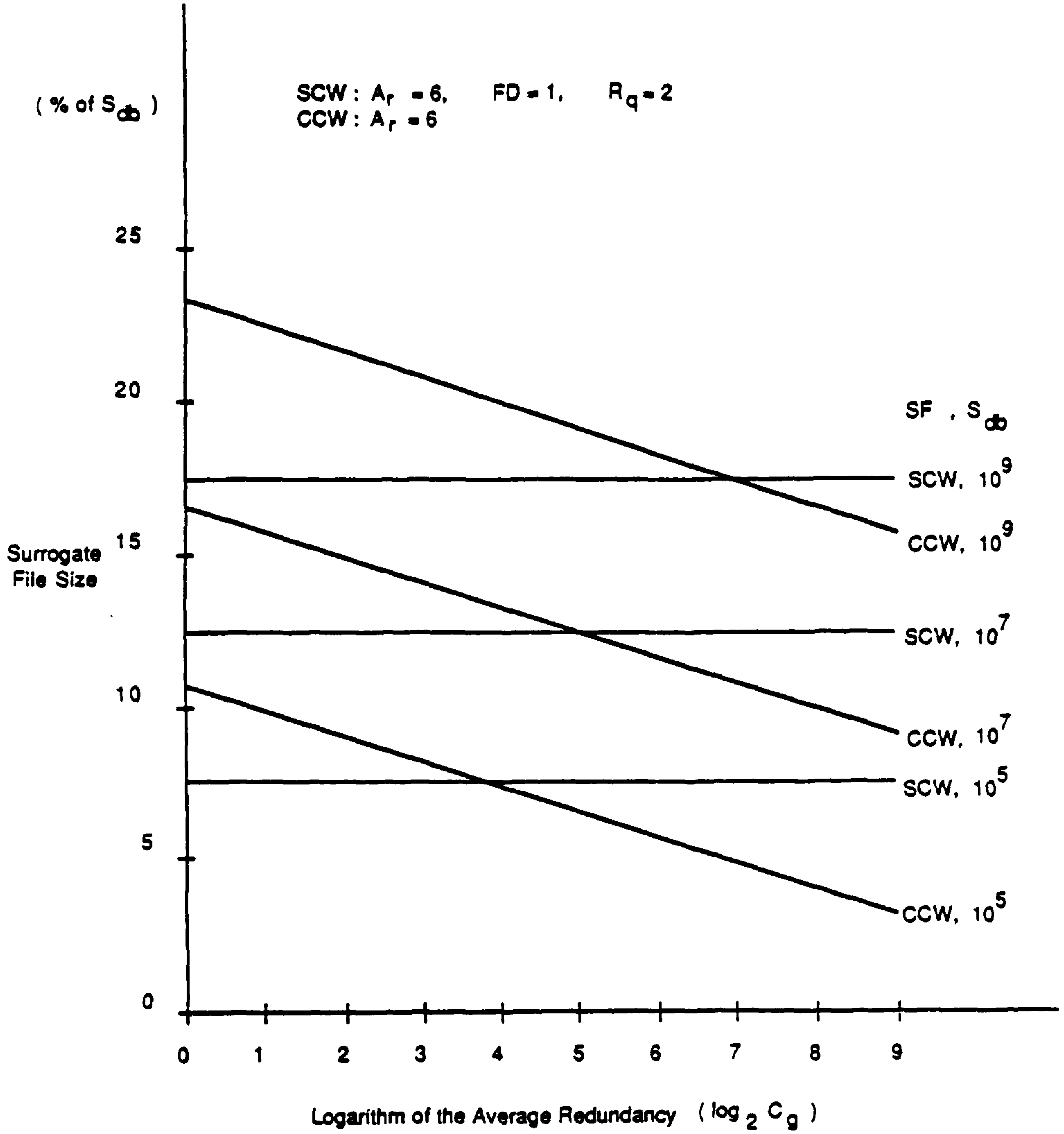


Figure 4.3 SCW and CCW Surrogate File Size Comparison

When the size of the EDB is less than 10^7 bytes, the surrogate file size is less than 2 Mbytes, so the whole surrogate file can be stored in a fast memory to speed up the retrieval process.

4.2. Query Response Time

For the query response time, we assumed that the hashing function is ideal, so there are no false drops with the CCW surrogate file technique and the SCW surrogate file technique has only the false drops caused by the logical OR operation on the BCW's. A partial-match query is assumed and the BCW of the surrogate file is compared with the QCW by using sequential byte by byte comparison. The query response time results for the SCW and CCW techniques are obtained from the equations developed in the previous section and are shown in Figures 4.4 through 4.12. Table 4.1 shows the values of parameters used in this simulation. The parameters relating to the disk are obtained from the characteristics of the DEC RA81 disk.[6]

In Figures 4.4 through 4.6 and 4.7 through 4.9, we plot the query response times, QT_{scw} and QT_{ccw}, and corresponding subprocessing times for S_{db} of 10^5, 10^7 , and 10^9 bytes, respectively. When S_{db} is 10^5 bytes, most of the query response time is spent for EDB access. But when S_{db} is 10^9 bytes, the query response time becomes very large and most of the query response time is spent for surrogate file accessing and searching because of the increased surrogate file size and sequential searching of the surrogate file. The number of arguments in a fact (A_r) has little affect on either QT_{scw} or QT_{ccw} since we assumed that the S_{db} remains constant under the variations in A_r.

Parameter	Value
Average seek time	28 msec
Minimum seek time	6 msec
Rotational delay	8.3 msec
Data transfer rate	2K bytes/msec
Data sector size	512 bytes
Sectors/track	52
Tracks/cylinder	7
Time for byte comparison	3 μsec
Unique id collection time	10 μsec
Block size	2K bytes

Table 4.1. The Values of Parameters Used in the Simulation

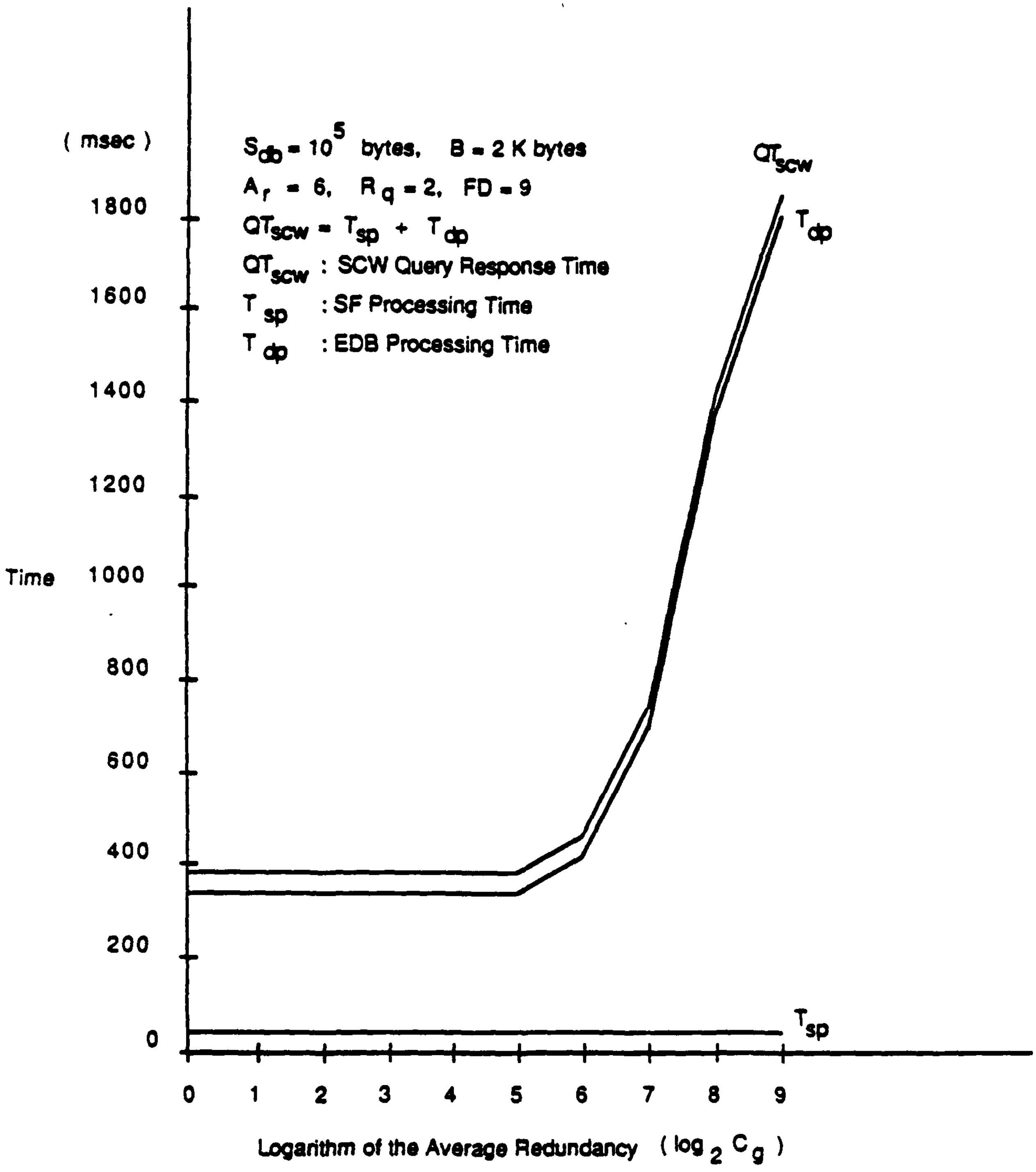


Figure 4.4 Components of the SCW Query Response Time ($S_{db} = 10^5$ bytes)

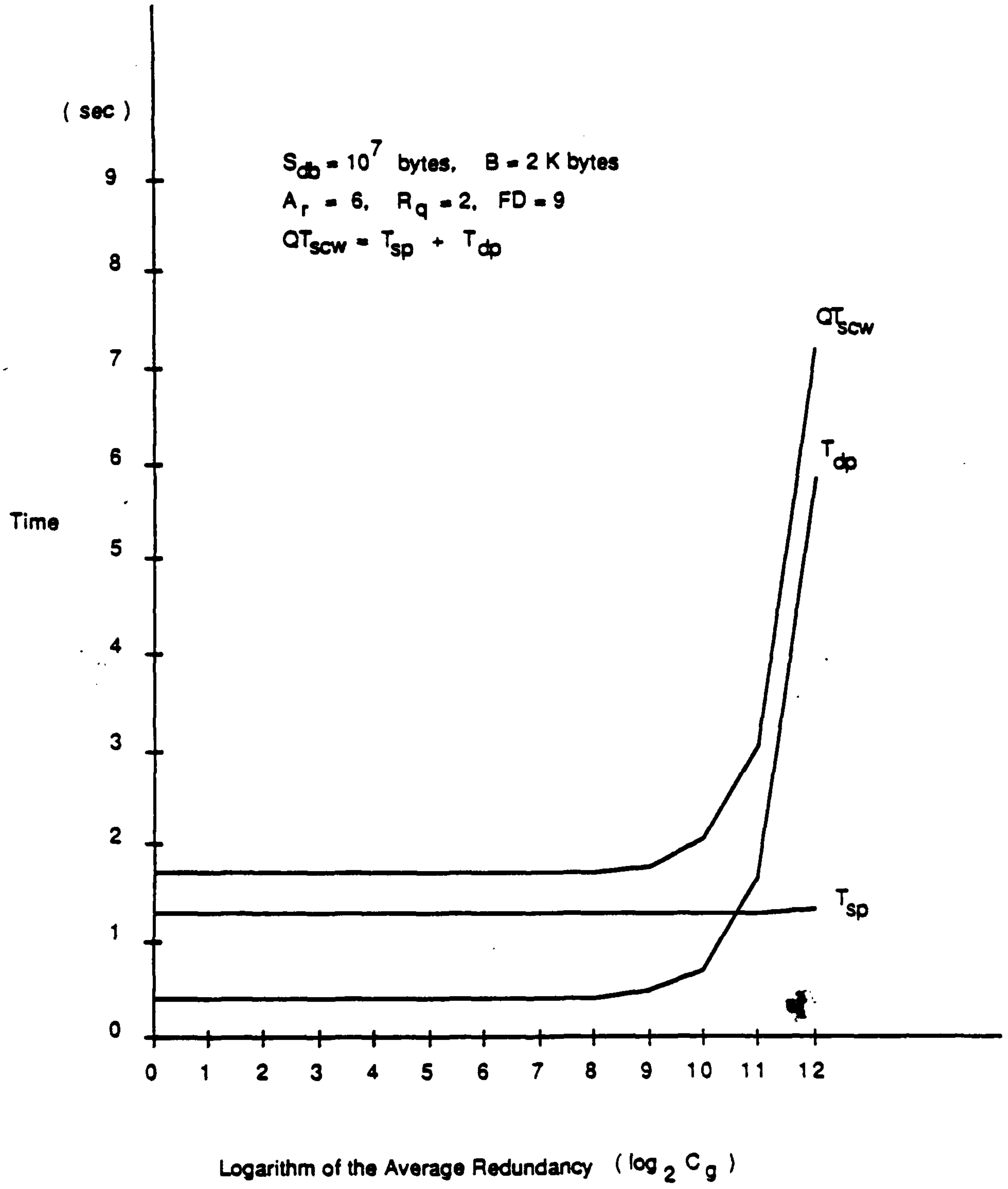


Figure 4.5 Components of the SCW Query Response Time ($S_{db} = 10^7$ bytes)

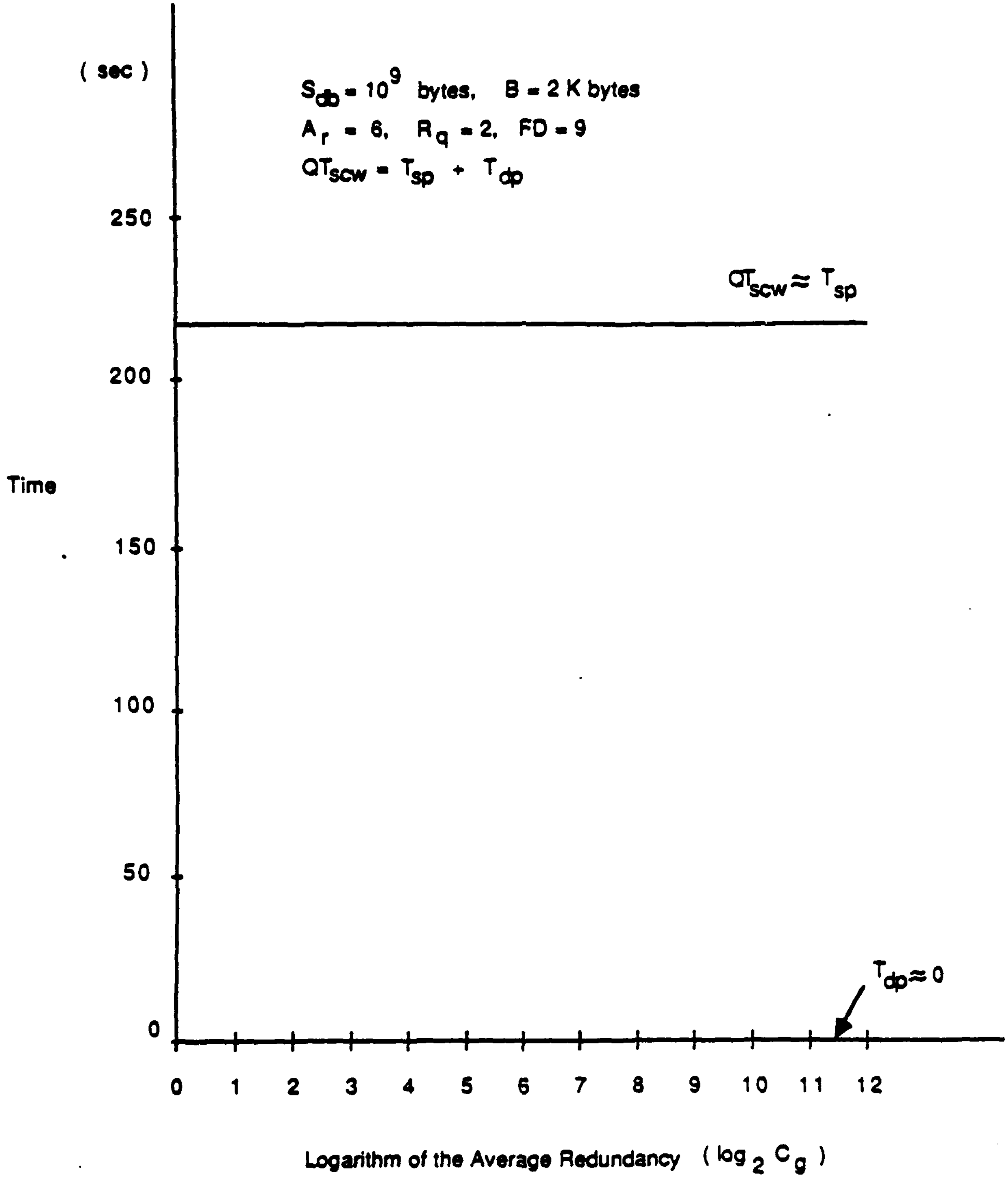


Figure 4.6 Components of the SCW Query Response Time ($S_{db} = 10^9$ bytes)

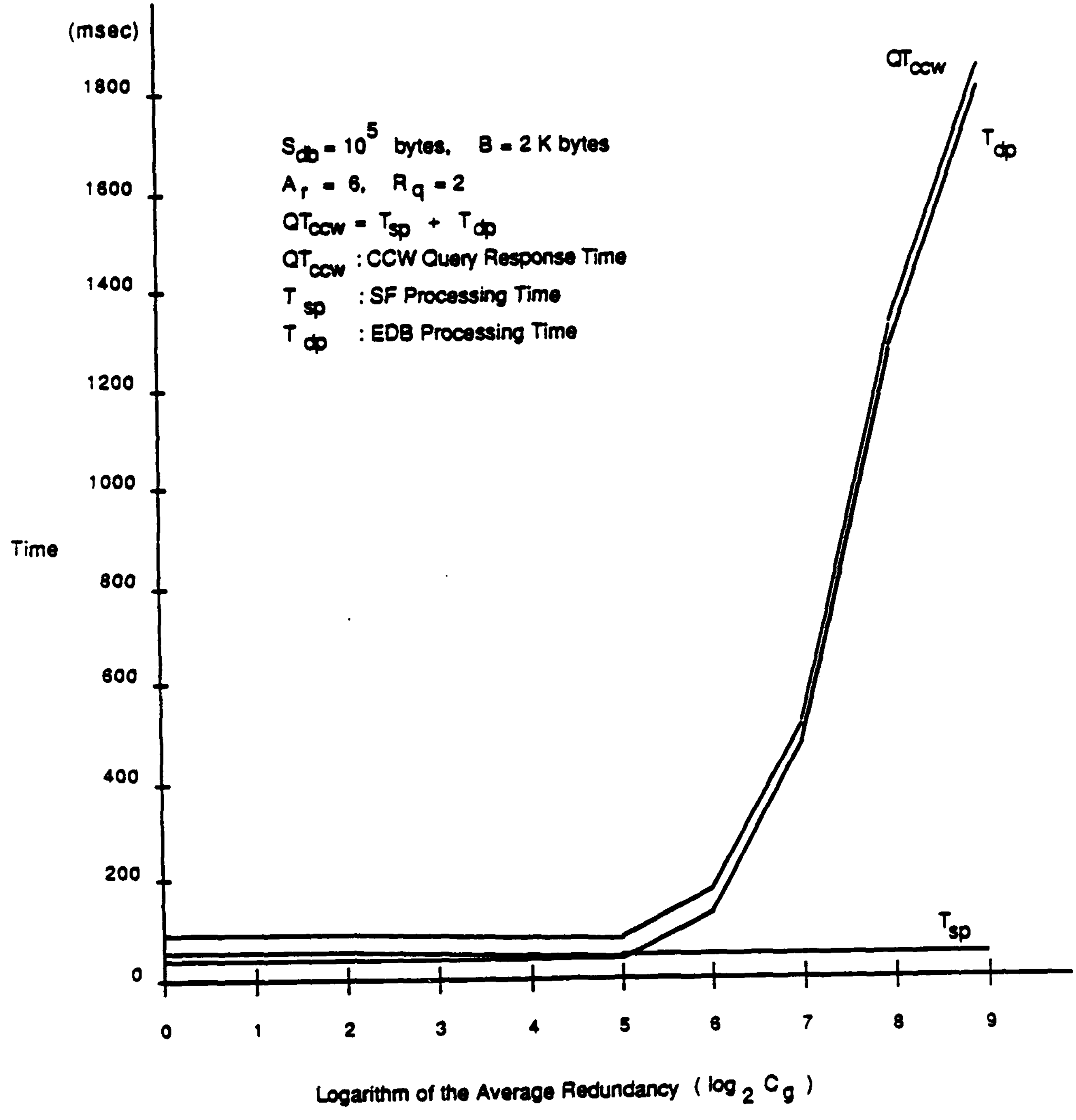


Figure 4.7 Components of the CCW Query Response Time ($S_{db} = 10^5$ bytes)

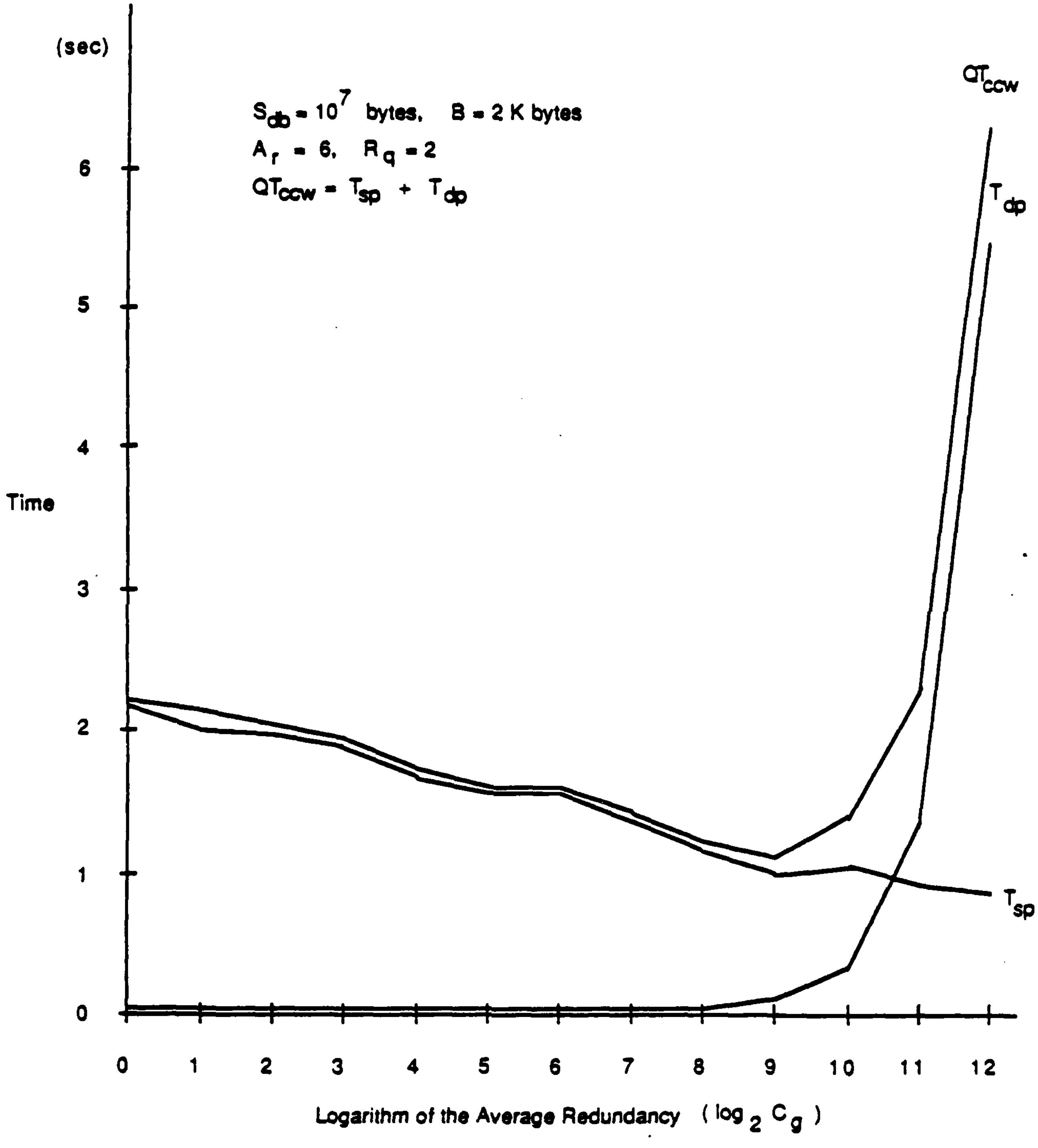


Figure 4.8 Components of the CCW Query Response Time ($S_{db} = 10^7$ bytes)

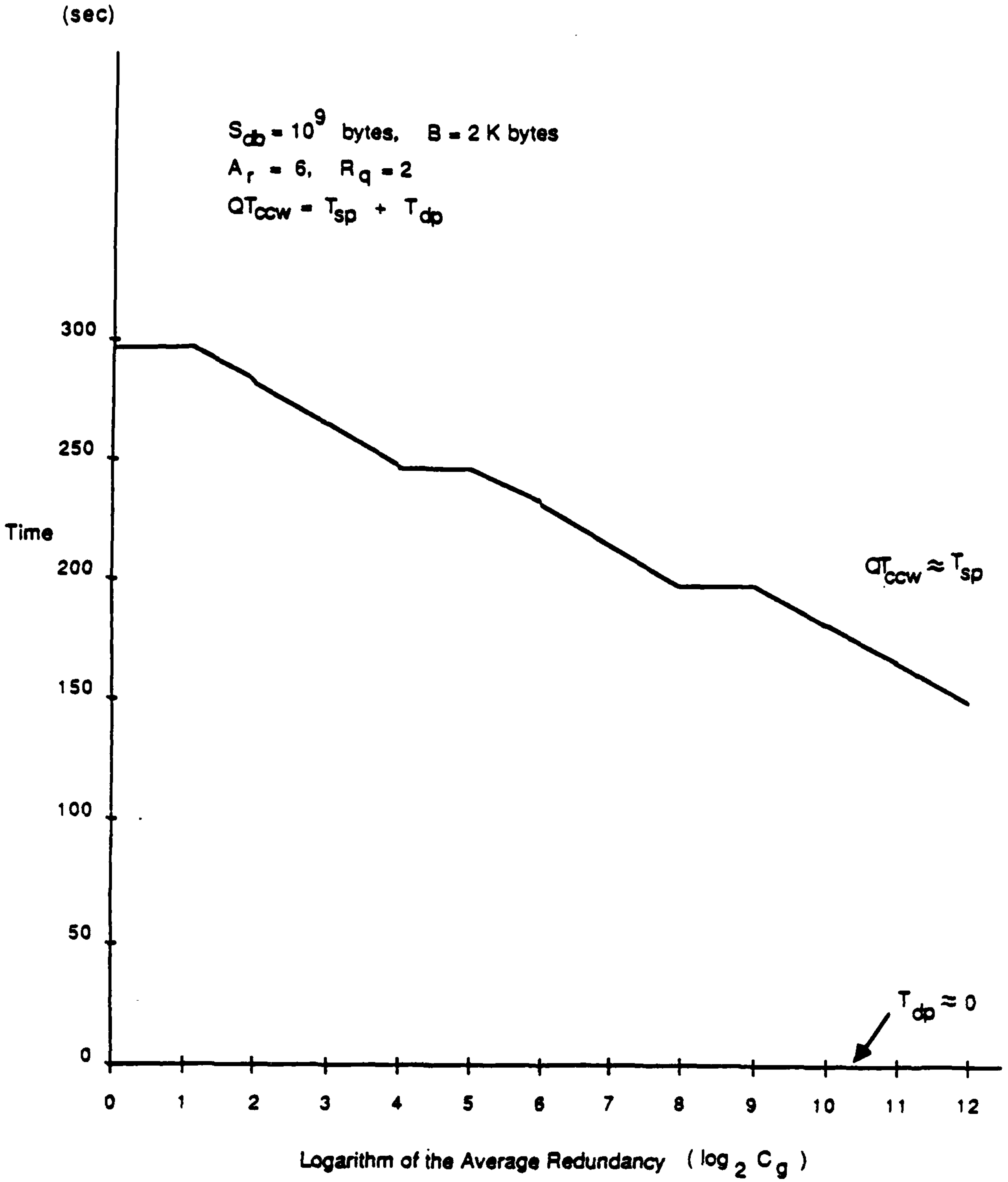


Figure 4.9 Components of the CCW Query Response Time ($S_{db} = 10^9$ bytes)

When S_{db} is 10^5 bytes, R_q is not a factor which affects QT_{scw}, but QT_{scw} increases as FD increases. However, when S_{db} is 10^9 bytes, the result is reversed, that is, R_q affects the QT_{scw} considerably while FD does not. There are two reasons supporting this result:

1) S_{scw} decreases as R_q increases. However, when S_{db} is small, S_{scw} is also small for any R_q so that the time for accessing and searching the SCW file is almost constant. Therefore, the time for accessing the EDB, which depends on FD, becomes a major factor in QT_{scw}.

2) When S_{db} is large, S_{scw} becomes large so that most of QT_{scw} is spent for accessing and searching the SCW file. Therefore, S_{scw} is a main factor deciding QT_{scw}. Since S_{scw} largely depends on R_q, the change in R_q is directly reflected in QT_{scw}.

QT_{scw} and QT_{ccw} are largely affected by C_g when S_{db} is 10^5 bytes and R_q is small. However, as R_q becomes large, the effect of C_g on QT_{scw} and QT_{ccw} decreases. This fact is well explained by the role of R_q and C_g in determining the number of good drops:

1) If R_q is small and C_g is large, then there are so many good drops that a large amount of time is required for accessing the EDB.

2) If R_q becomes large, the number of good drops decrease considerably, and so does the EDB access time, which is the major component of the query response time when S_{db} is 10^5 bytes.

From Figures 4.6 and 4.9, we can see that when S_{db} is 10^9 bytes, as C_g increases, QT_{scw} remains constant while QT_{ccw} decreases. This occurs because a fewer number of bits is required to uniquely identify each attribute value in the CCW case. But when C_g is larger than a certain value, the query response time starts increasing because of the increased EDB access time. Also, we can see from Figures 4.6 and 4.9 that most of the query response time is used for the surrogate file accessing and searching when the EDB is large. Therefore, if we use multiple processors and/or associative memory to speed up the surrogate file processing, we can reduce the query response time considerably. Since the surrogate files are quite regular and compact, they can be mapped into the associative memory. Thus, we can obtain a speed up by the content addressing capability and the parallelism of the associative memory.[1, 2] In addition, we can also obtain a speed up proportional to the number of processors because there is little need for communication among the processors.

Since searching and disk access can be overlapped, if we increase the block size, then the number of disk accesses can be reduced and we can save time as long as the block searching time is less than the block access time. In the case of a multiple disk system, the surrogate file and the EDB are distributed over a number of disks and we can reduce the disk access time by seeking several disks concurrently.

To compare the retrieval performance of the SCW and CCW techniques, we plot QT_{scw} and QT_{ccw} in Figures 4.10 through 4.12 for various parameter values. From those figures we can see that QT_{ccw} is smaller than QT_{scw} when R_q is small. because S_{ccw} is smaller than S_{scw} when R_q is small.

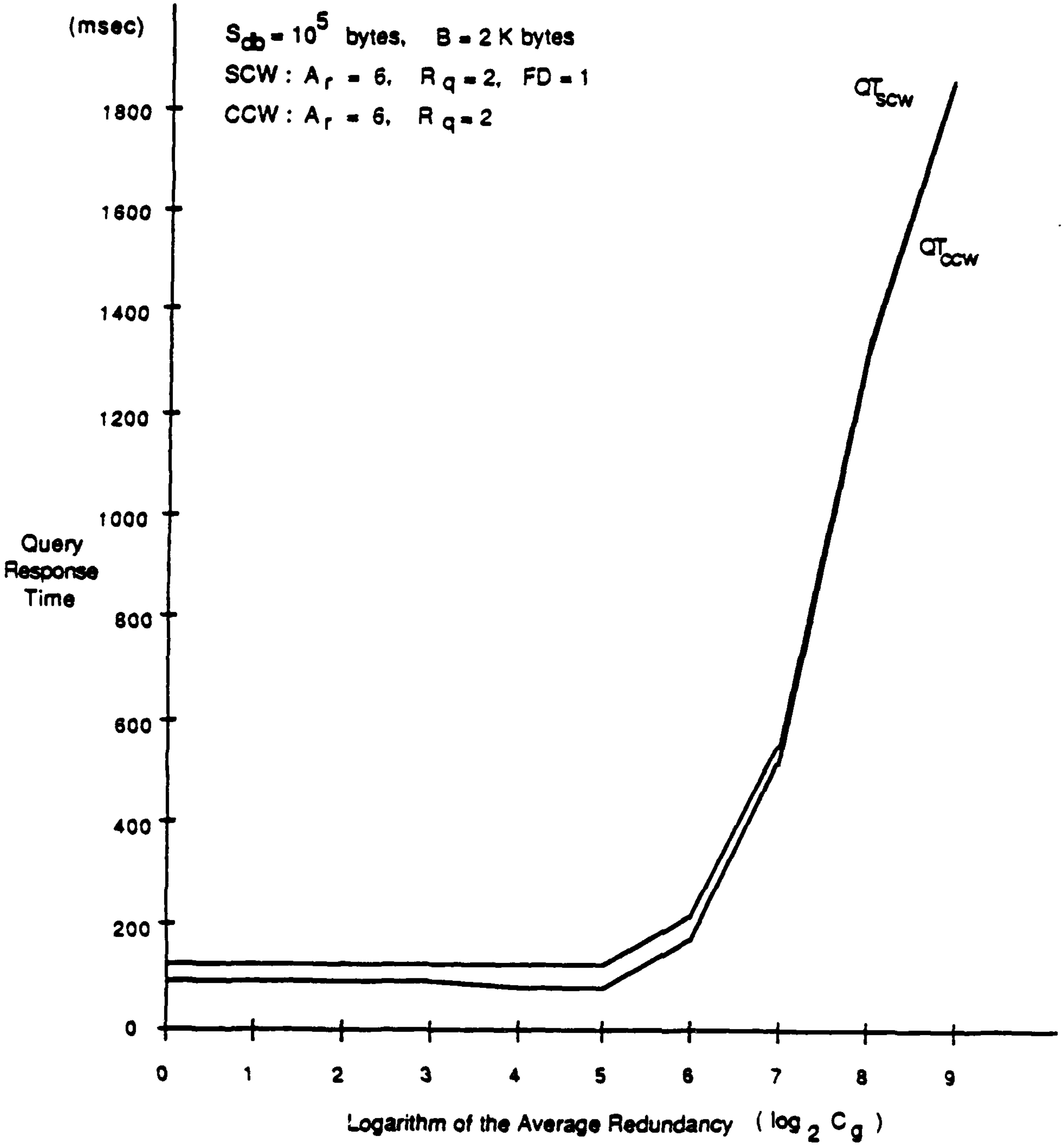


Figure 4.10 SCW and CCW Query Response Time Comparison ($S_{db} = 10^5$ bytes)

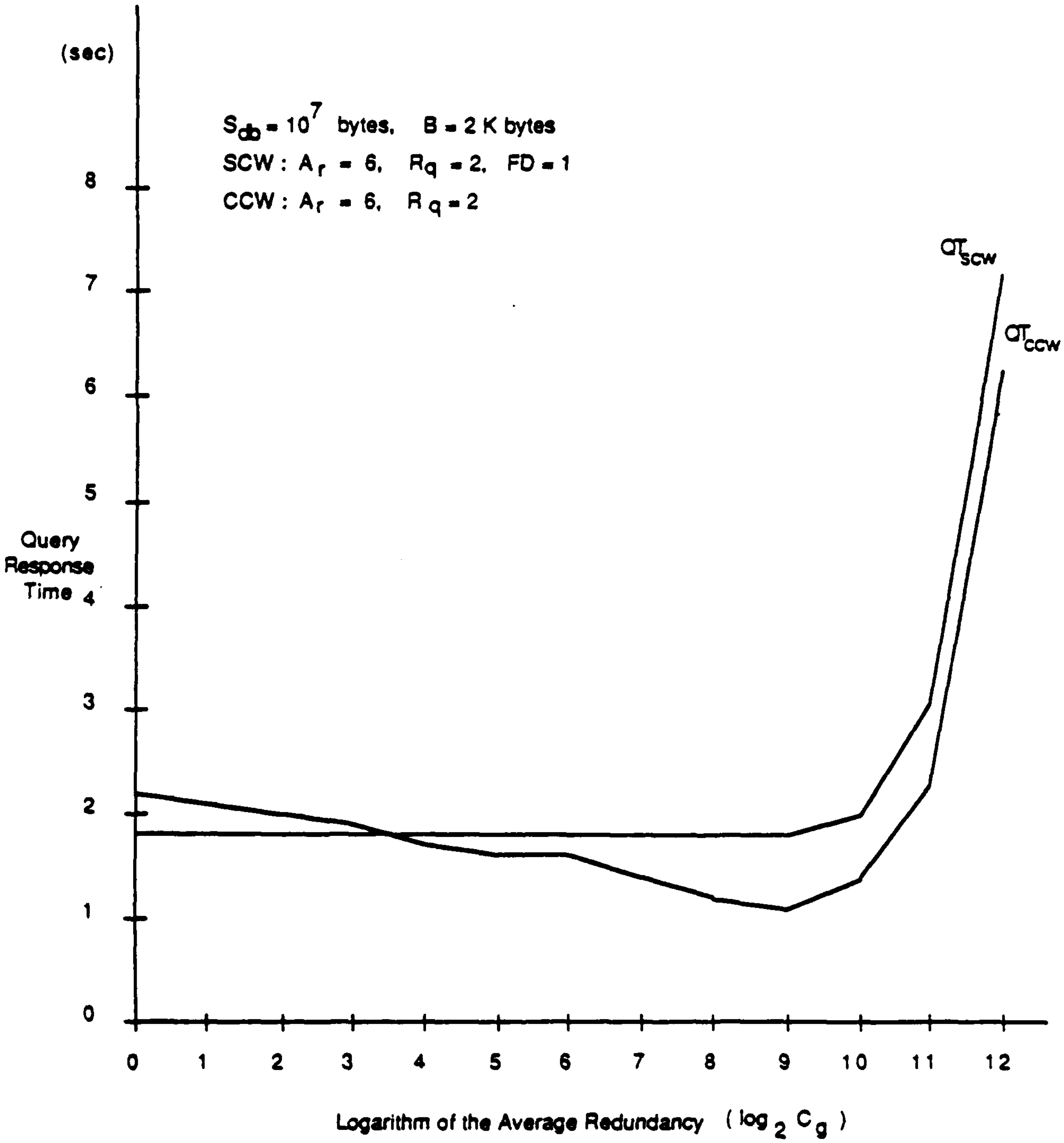


Figure 4.11 SCW and CCW Query Response Time Comparison ($S_{db} = 10^7$ bytes)

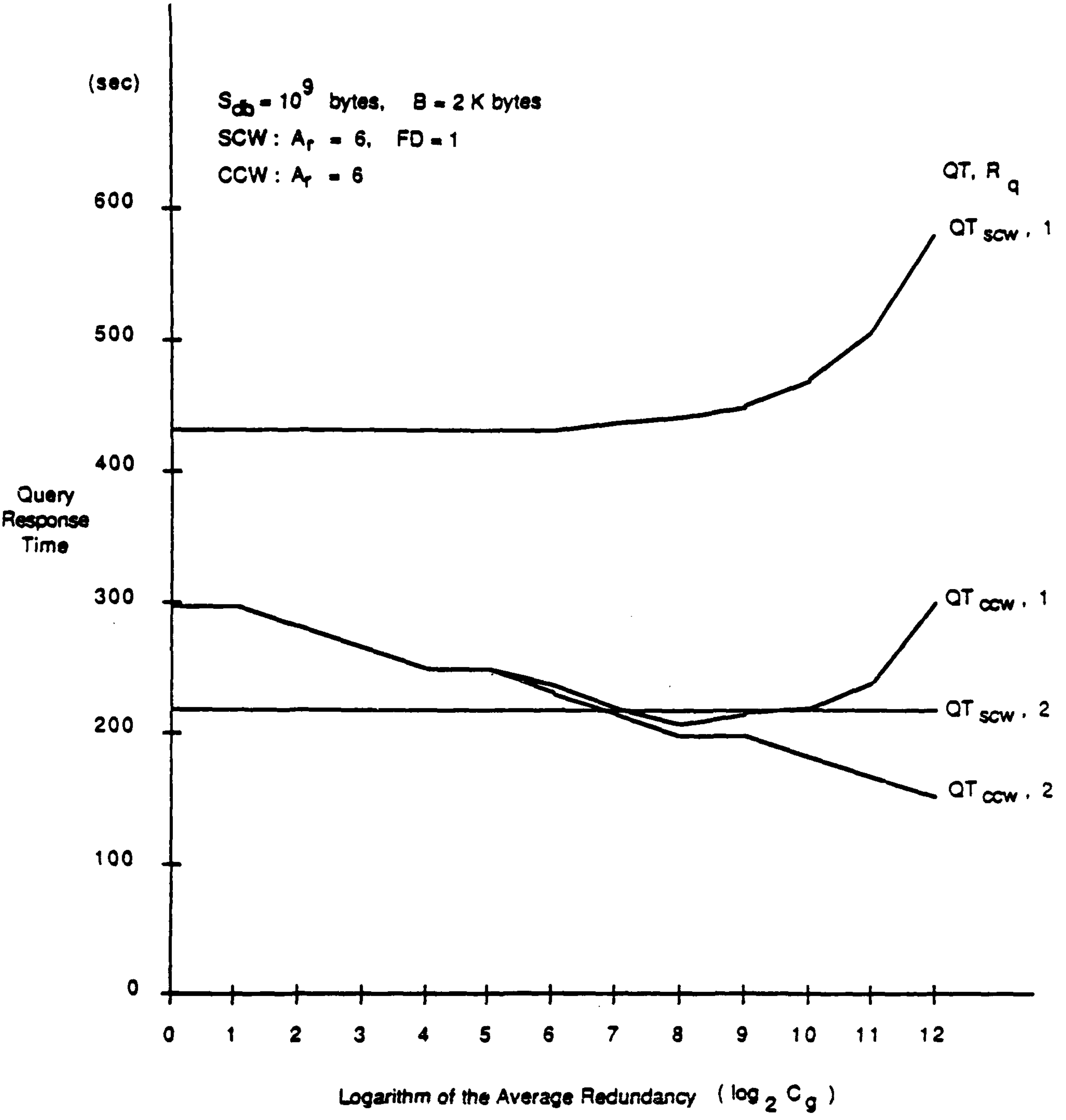


Figure 4.12 SCW and CCW Query Response Time Comparison ($S_{db} = 10^9$ bytes)

5. Comparison of SCW and CCW Surrogate File Techniques

The size and query response time of the CCW is smaller than those of the SCW when the average number of arguments specified in a query is small.

It is very easy to update SCW or CCW surrogate files. When a new fact is added to the EDB, the corresponding code word is simply appended to the existing SCW or CCW surrogate files. No other operations are required. To delete a fact, we must find and delete the entry in the surrogate file as well as in the EDB. When one changes the value of a field, SCW requires that a new code word be generated and the old one deleted. For CCW the change need only be made to the portion of the code word in question.

One obvious advantage of CCW over the SCW is that many relational operations can be easily performed on the CCW surrogate file rather than on the relations themselves.[2] This offers considerable potential savings in time to carry out those relational operations.

In SCW, the order of argument positions in either query or fact can't be differentiated because a SCW is generated by the logical OR operations on the BCW's. This property of SCW can be a disadvantage when used for rule indexing in the context of logic programming.

SCW surrogate file searching time can be reduced by using the bit-sliced organization to store the SCW files.[14] But in that case, we must read and write back many blocks of SCW surrogate file to update one SCW, which is not tolerable when the EDB is dynamic.

In the SCW surrogate file technique, to reduce the the inherent false drops caused by the logical OR operations on the BCW's, one may assign different code weights to the BCW's of argument values depending on the occurrence frequency and query frequency of the argument values. But to do this, the code weights of frequently occurring argument values must be maintained in a table to be looked up whenever generating a binary code word. [9, 18]

6. Further Research Consideration

The main drawback of the SCW and CCW surrogate file technique is that the whole surrogate file must be read to the main memory and searched. To reduce the searching time, one can produce a block code word for each block of the surrogate file and use the block code words as an index for the surrogate file. A given QCW is compared with the block code words first and only those blocks of the surrogate file whose corresponding block code words match the QCW are retrieved and searched. But the speed up is achieved at the expense of the extra storage space and maintenance cost for the block code words. The performance of the block code words will depend on the following factors:

1) Type of hashing functions used for code generation

2) Algorithm for generating the block code words.

3) Blocking factor: number of code words blocked together to form a block code word.

4) How frequently the database will change.

J. L. Pfaltz introduced the block descriptor generated by logical Oring the disjoint codes of each record [16] and R. Sacks-Davis and K. Ramamohanarao developed two level superimposed coding scheme.[19, 20]

It has been shown that surrogate file processing time is dominant when the EDB is very large. Thus, if we adopt multiple processors and/or associative memory, we can reduce the surrogate file processing time considerably. A general structure of a back end system which contains multiple processors for the management of a very large extensional database of facts is shown in Figure 6.1. We assume that there are gigabytes of data stored on the EDB disks and there are gigabytes of CCW surrogate files stored on the SF disks. Suppose that the user is interested in retrieving fact data given some subset of values from a particular relation. The query code word would be constructed in the Request Processor using the proper hashing function and considering the positions of the values within the relation. The QCW would then be broadcast to all of the Surrogate File Processors (SFP's) to be used as a search argument. One could think of the SFP as a processor with associative memory with the QCW as the search argument. The SFP compares the QCW with each CCW and *strip off* the unique identifiers of matching CCW's. As soon as any unique identifiers are found by the SFP's they can be sent to the collector and passed on to the Extensional Data Base Manager (EDBM) for processing. The EDBM will retrieve the facts, compare them with the query to insure that a false drop has not occurred, put them in blocks, and send the blocks to the logic programming engine.

Furthermore, the SFP's can be extended to support complex relational algebra operations such as join. Consider a join using the hash join algorithm. [3, 10, 23] Since the surrogate files already consist of hash values, we only need to partition the portion of code words that represent the join variable and the associated unique identifiers into buckets according to the ranges of code words. Then, based on matching within each bucket (which can be done in parallel), pairs of unique identifiers can be sent to the EDBM for final verification.

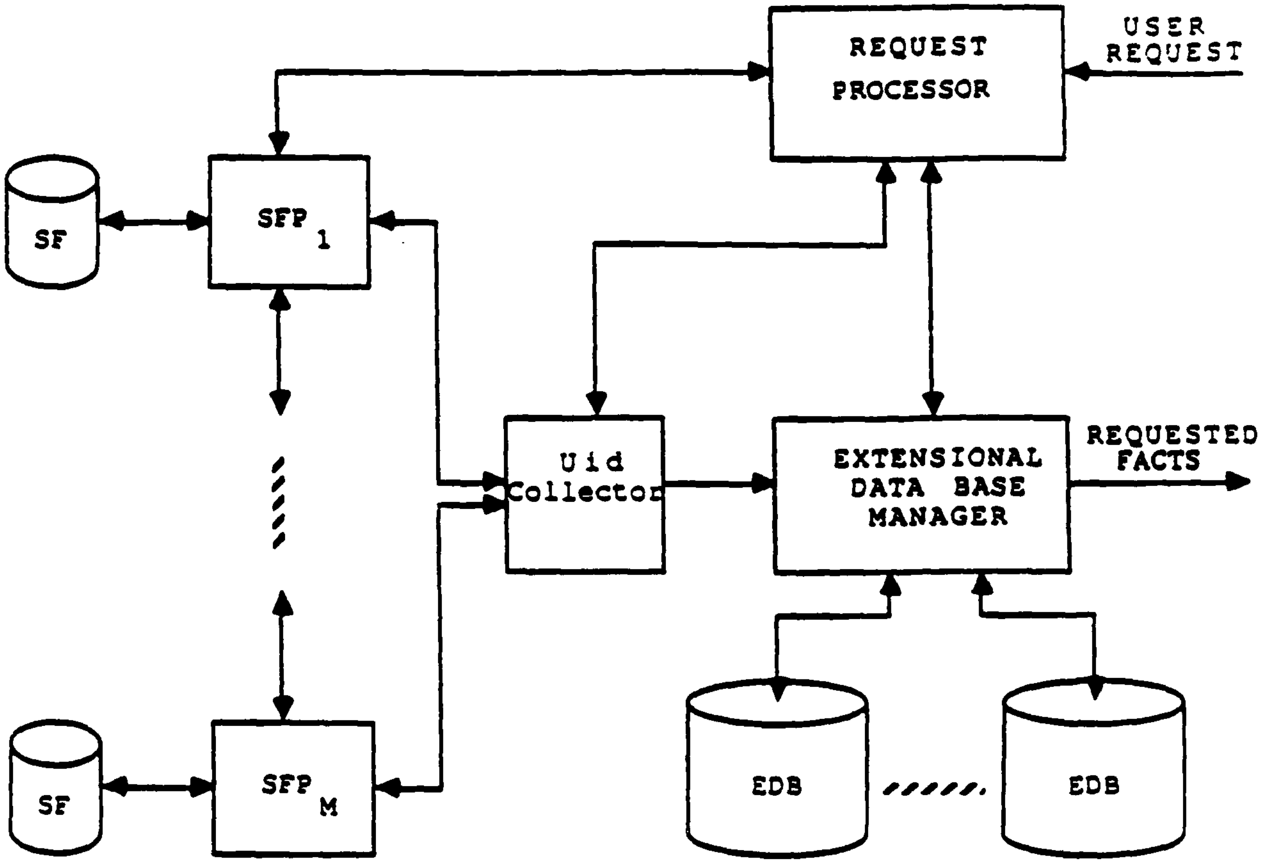


Figure 6.1 Back End System for Fact Management

7. Conclusion

CCW and SCW surrogate file techniques are analysed in terms of storage requirements and retrieval performance. The size and query response time of the CCW is smaller than those of the SCW when the average number of arguments specified in a query is small. Since the size of the CCW and SCW files are generally less than 20% of the EDB and the maintenance of those files is very simple, they are suitable for the applications requiring very large dynamic EDB. Additionally, many relational operations can be performed on the CCW surrogate files rather than on the relations.

CCW and SCW surrogate file techniques can be implemented easily with multiple processors and/or associative memory to speed up the retrieval process in very large knowledge base system. Our future research is towards the development of special architectures supporting those surrogate file techniques.

References

[1] S. R. Ahuja, C. S. Roberts, "An Associative/Parallel Processor for Partial Match Retrieval Using Superimposed Codes," Proc. 7th Annual Symp. on Computer Architecture, May 1980, pp.218-227.

[2] P. B. Berra, S. M. Chung, N. I. Hachem, " Computer Architecture for a Surrogate File to a Very Large Data/Knowledge Base," IEEE Computer Vol. 20, No.3, March 1987, pp.25-32.

[3] K. Bratbergsengen, " Hashing Methods and Relational Algebra Operations," Proc. VLDB, 1984, pp.323-333.

[4] A. F. Cardenas, " Analysis and Performance of Inverted Data Base Structures," CACM, Vol. 18, No. 5, 1975, pp.253-263.

[5] R. M. Colomb, " A Hardware-Intended Implementation of Prolog Featuring A General Solution to the Clause Indexing Problem," Ph.D dissertation, Univ. of New South Wales, Australia, 1986.

[6] Digital Equipment Corporation, RA 81 Disk Drive User Guide, 1982

[7] H. C. Du, S. Ghanta, et al.," An Efficient File structure for Document Retrieval in the Automated Office Environment," Proc. Int'l Conf. on Data Engineering, 1987, pp.165-172.

[8] C. Faloutsos, S. Christodoulakis, " Signature Files: An Access Method for Documents and Its Analytical Performance Evaluation," ACM Trans. on Office Information Systems, Vol. 2, No. 4, 1984, pp.267-288.

[9] C. Faloutsos, S. Christodoulakis, " Design of a Signature File Method that Accounts for Non-Uniform Occurrence and Query Frequencies," Proc. VLDB, 1985, pp.165-170.

[10] M. Kitsuregawa, H. Tanaka, T. Moto-Oka, " Application of Hash to Data Base Machine and Its Architecture," New Generation Computing, Vol. 1, 1983, pp.63-74.

[11] P. -A. Larson, " Performance Analysis of Linear Hashing with Partial Expansions," ACM Trans. on Database Systems, Vol. 7, No. 4, 1982, pp.566-587.

[12] P. -A. Larson, A. Kajla, " File Organization: Implementation of a method Guaranteeing Retrieval in One Access," CACM, Vol. 27, No. 7, 1984, pp.670-677.

[13] P. -A. Larson, " Hash Files: Some Recent Developments," Proc. 1st Int'l Conf. on Supercomputing Systems, 1985, pp.671-679.

[14] D. L. Lee, " A Word-Parallel, Bit-Serial Signature Processor for Superimposed Coding," Proc. Int'l Conf. on Data Engineering, 1986. pp.352-359.

[15] J. Martin, Computer Data Base Organization, second edition, Prentice-Hall, 1977

[16] J. L. Pfaltz, W.J. Berman, and E.M. Cagley, " Partial-Match Retrieval Using Indexed Descriptor Files," CACM, Vol. 23, No. 9, 1980, pp.522-528.

[17] K. Ramamohanarao, J. Shepherd, " A Superimposed Codeword Indexing Scheme for Very Large Prolog Databases," Proc. 3rd Int'l Logic Programming Conference, 1986, pp.569-576.

[18] C.S. Roberts, " Partial Match Retrieval via the Method of Superimposed Codes," Proceedings of the IEEE, Vol. 67, No. 12, 1979, pp.1624-1642.

[19] R. Sacks-Davis, K. Ramamohanarao, "A Two level Superimposed Coding Scheme for Partial Match Retrieval," Information Systems Vol. 8, No. 4, 1983, pp.273-280.

[20] R. Sacks-Davis, " Performance of a Multi-key Access Method Based on Descriptors and Superimposed Coding Techniques," Information Systems, Vol. 10, No. 4, 1985, pp.391-403.

[21] D. Shin, P.B. Berra, " An Architecture for Very Large Rule Bases Based on Surrogate Files," Proc. 5th Int'l Workshop on Database Machines, 1987, pp.555-568.

[22] D. Tsichritzis, S. Christodoulakis, " Message Files," ACM Trans. on Office Information Systems, Vol. 1, No. 1, 1983, pp.88-98.

[23] P. Valduriez, G. Gardarin, " Join and Semijoin Algorithms for a Multiprocessor Database Machine," ACM Trans. on Database Systems, Vol. 9, No. 1, 1984, pp.133-161.

[24] M. Wada, Y. Morita, et al., " A Superimposed Code Scheme for Deductive Databases," Proc. 5th Int'l Workshop on Database Machines, 1987, pp.569-582.

[25] M. J. Wise, David M. W. Powers, " Indexing Prolog Clauses via Superimposed Code Words and Field Encoded Words," Proc. Symposium on Logic Programming, 1984, pp.203-210.

Appendix 9-B

Back End Architecture based on Transformed Inverted Lists, A Surrogate File Structure for a Very Large Data/Knowledge Base

Nabil I. Hachem *P. Bruce Berra*

Department of Electrical and Computer Engineering,
Syracuse University, Syracuse, NY 13244-1240

ABSTRACT

Knowledge based systems have gained prominence in the rapidly growing field of Artificial Intelligence (AI). The current state of the art of such systems focuses on narrow domains of knowledge bases with limited applications. In the future, those systems must deal with more general applications and existing Very Large Databases present a source of information to be used for these AI applications. Surrogate file techniques, which rely on a compressed image of the database, present a promising approach to the formidable management task of such Very Large Data/Knowledge bases. In this paper we present a detailed analysis of Transformed Inverted Lists (TIL), an inverted surrogate file structure, and describe a parallel back end architecture, based on TIL, for the management of a Very Large Data/Knowledge Base.

Index Terms: Very Large Data/Knowledge Base, Surrogate File, Inverted Lists, Indexed Files, Database Machines.

1. Introduction

Knowledge based systems have gained prominence in the rapidly growing field of Artificial Intelligence (AI). The current state of the art of such systems focuses on narrow domains with limited applications. Existing Very Large Databases present a rich source of information to be used for AI applications and new demands for the management of such Data/Knowledge bases are foreseen to be essential for the new generation of knowledge based systems. Berra et al [BER87] relate the problem to the general partial match retrieval problem and various techniques have been studied for partial match retrieval using surrogate files.

The term surrogate file (SF) dates back to early work in information retrieval and other equivalent terms such as signature and descriptor files are also used for such structures [1]. Typical work related to surrogate files processing is found with the Superimposed Coding method of Roberts [ROB79]. Ahuja et al [AHU80] and Lee [LEE86] proposed associative architectures for the fast processing of superimposed code words. Furthermore, Colomb et al [COL86] and Wise et al [WIS84] relate the technique as an indexing scheme for a logic programming environment. Lloyd et al [LLO80 and 82] have taken an interresting approach in which they select bit values in the facts and interlace them to form a code word.

The approach presented in this paper relies on an inverted list indexing scheme that is performed on the surrogate files instead of the usual database inversion applied in conventional information retrieval systems. In [BER87], Concatenated Coding and Transformed Inverted Lists (TIL) were introduced as efficient surrogate file techniques for the management of Very Large Knowledge bases. Our scope, in this paper, is to present a deterministic analysis of the TIL technique and introduce a parallel back end system for the management of Very Large Knowledge bases.

We begin by introducing the system model in Section 2, then derive the minimum storage and query response time equations in Sections 3 and 4. In Section 5, simulation results are presented followed by a discussion of the maintenance aspects of the TIL technique in Section 6. Section 7 introduces a parallel architecture for the processing of the TIL surrogate files. Finally, a summary of our results with some concluding remarks are given in Section 8.

2. System Model for Transformed Inverted Lists

Single or multilevel indexing is a common technique used in database management systems (DBMS) for fast data access. In partial match retrieval, creating index files for more than one field in a record is necessary. The extreme case arises when every entry in a record is indexed independently and is referred to as inverted lists organization [DAT86, Chap. 21]. The problem behind using inverted lists is that the size of the indices can become enormous, equal to or even larger than the database size.

Transformed Inverted Lists (TIL) are similar to inverted lists with the main difference that indices are built based on the binary representation (BR) of the hashed output of a given field in a record of the database relation. Two TIL types, TIL1 and TIL2, are considered in this paper. A simple relation is illustrated in Figure 2.1. The fields are referred to as arguments and the BR values for argument position 2 are listed.

The application environment of the TIL technique would be the management of a large Knowledge Base of facts, referred to as the extensional database (EDB), within a logic programming context. We assume that many different relations (fact types) with varying degrees and cardinalities exist in the very large extensional database that we are considering. Furthermore, we assume that the tuples are stored in such a way that one first accesses the relation followed by an access to a particular tuple via its unique identifier (Uid). The unique identifier could be derived from the "primary

[1] Refer to Faloustos [FAL85] for a review of various methods for accessing textual data using surrogate files

Uid	Ar_1	Ar_2	Ar_3	Ar_4
uid1	.....	br1	.....	.....
uid2		br2		
uid3		br3		
uid4		br1		
uid5		br4		
uid6		br1		
uid7		br5		
uid8		br6		
uid9		br6		
uid10		br7		
uid11		br3		
uid12		br4		

Figure 2.1 A Simple Knowledge Base Relation

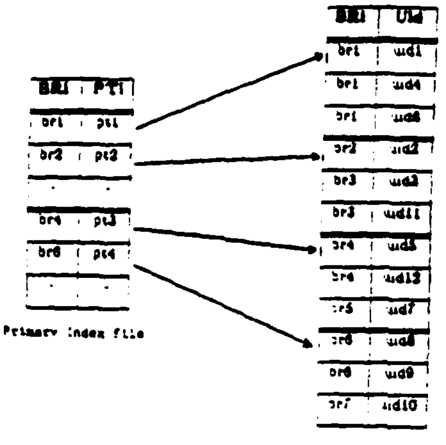


Figure 2.2 TIL1 for Ar_2 in Figure 2.1

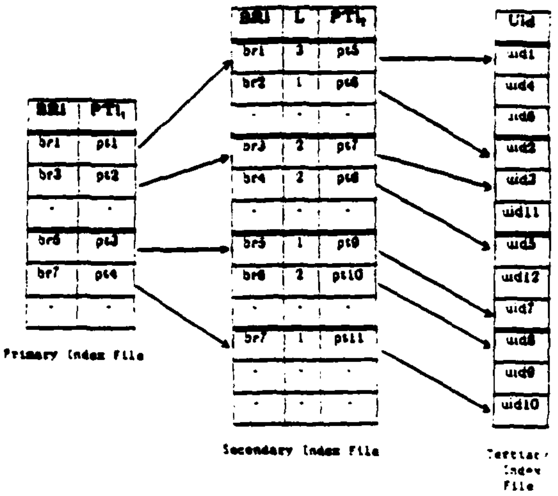


Figure 2.3 TIL2 for Ar_2 in Figure 2.1

key" of the relation or a serially generated number attached to each fact. We will obtain the n .me of the relation and a subset of values along with their positions in the relation from the logic programming process when it requests data. Thus, the storage structure for the actual facts themselves would be very simple and a method such as extendible hashing (Fagin FAG79]) could be used to guarantee retrieval of a given fact in at most two disk accesses. This presupposes that all secondary key retrievals will take place on the surrogate file or through post processing of the retrieved tuples if there are many different types of users of the same database.

2.1. TIL1 Description

TIL1 consists of a two level indexed inverted list. Figure 2.2 illustrates the TIL1 organization for argument position 2 of the relation of Figure 2.1. The blank entries in the primary index file are usually included for updating purposes. The secondary index file for a given argument in a tuple is an ordered list of the BRs of the hashing function output of that argument with the attached unique identifier (Uid). The first entry in each block of this file is duplicated in the primary index file with an attached pointer to the corresponding secondary index block address. Furthermore, index files are partitioned in blocks of B bytes each. It is observed that the entries in the primary index file are ordered as well.

When a given BR is to be retrieved (say BR=br3), the primary index file is sequentially accessed using the BR as the search argument and the pointer to the secondary block address corresponding to that BR retrieved (pt2 in our example). Then the secondary file is accessed in a direct mode and the required block(s) retrieved and searched sequentially for the occurrence(s) of the requested BR. The output is a list of Uids (uid3 and uid11 for our example) corresponding to the value of the request.

2.2. TIL2 Description

TIL2 is a three level indexed inverted list organization and is illustrated in Figure 2.3 for the same example relation. The difference between TIL2 and TIL1 lies in that the TIL1 secondary index file is now split into two files: the TIL2 secondary index file and the tertiary index file. Each entry in the tertiary index file consists of a Uid, so that the number of entries in this file is equal to the number of records in the database relation. Each entry in the TIL2 secondary index file consists of three fields: the BR of the hashed function output of an argument value (say BR=br6), a list length entry "L" that provides the number of records in the database that have the same entry value in a given argument position (2 for br6) and a pointer to the address of the first Uid in the tertiary file that has BR=br6. This pointer consists of the block address and a displacement value in the block.

The retrieval process for TIL2 is similar to TIL1, but requires the access of an additional index level.

2.3. Partial Match on Multiple Argument Positions

When more than one argument position match is requested in a query, the different outputs from the inverted lists searches need to be intersected. The outcome of the intersection is a set of Uids that complies with the query requirements. Finally this set of Uids is used to directly access the main database for the retrieval of the matched records. The gain in retrieval time when using transformed inverted lists is mainly due to the small size of the surrogate

Notations	Meanings	Simulation Values
B	Size of a block (Blocking Factor)	2-16 Kbytes
T_{seek}	Disk seek time	28 msec
T_{lat}	Rotational latency	8 5 msec
TR	Disk transfer rate	2 Mbytes sec
WL	Processor word length	8 bits
T_w	Average word comparison time	3 μ sec
A_r	Number of arguments in a tuple	2-10
R_q	Average number of arguments in a query	
N	Number of tupies in the database	
DG	Average number of good drops	
S_{db}	Database size	10^5-10^9 bytes
$S_{TIL1orTIL2}$	TIL1 or TIL2 Surrogate File size	
$S_{index1,2or3}$	Primary. Secondary or Tertiary Index File Size	
BR	Binary representation of the hashing function output	
QT	Query Response Time	
SFT	Surrogate File processing time	
IT	Intersection time	
DA	Time for retrieving records in the database	
C_i	Value distribution factor. that is. the average number of records which have the same value in the i-th argument	
C_g	Average of value distribution factor (Avèrage redundancy)	1-4096

Table 2 1 Summary of Notations

files and the fast access resulting from the indexing scheme Only conjunctive partial match queries are considered. but the reader should be aware that disjunctive queries have the same level of complexity. with the lists intersection operation replaced by a multiple sets union operation

It is noted that the inversion level of the surrogate files is determined by the AI application being considered. Since our underlying application involve logic programming and relational databases. we will assume fully inverted surrogate files throughout. In Sections 3 and 4. we derive the minimum storage and the query response time equations. The analysis is based on a compact representation of the data and does not take into account overflow chains It is meant to pinpoint performance bottlenecks. to be resolved in the design of the back end system. Table 2.1 lists the main abbreviations used in our analysis Due to space limitations. we present the equations and simulation results for TIL1 only and refer the reader to Hachem et al HAC87 for additional details.

3. Minimum Storage Overhead for TIL Surrogate Files

In this section the minimal sizes of TIL1 files are derived assuming no blank entries are available in the index files Those sizes are based upon the following parameters:

1) The bit length of the hashing function output for an argument in a record. denoted BR. must be at least $\left\lceil \log_2 \frac{N}{C_i} \right\rceil$ bits where C_i. called the value distribution factor. is the average number of records whose i-th arguments have the same value

2) The Unique Identifier (Uid) for each tuple is encoded in $\lceil \log_2 N \rceil$ bits. In practical applications. the Uids are encoded in a fixed number of bytes equal to $\left\lceil \frac{\lceil \log_2 N \rceil}{8} \right\rceil$.

Denoting by S_{index1} and S_{index2} the minimal sizes of the primary and secondary index files of TIL1 respectively the minimal size of TIL1 Surrogate Files is:

$$S_{TIL1} = S_{index2} + S_{index1}$$

3.1. Secondary Index File Size

Each entry in a secondary index file consists of two fields: 1) The Binary Representation (BR) of the hashing function output for an argument. 2) The Uid of the tuple in which it is found. The number of entries in the secondary index file. for each argument. is equal to the number of tuples N in the database. Therefore. the secondary index size (in bits) is given by:

$$S_{index2} = \sum_{i=1}^{A_r} S_{I2i} = \sum_{i=1}^{A_r} \left(\left\lceil \log_2 \frac{N}{C_i} \right\rceil + \lceil \log_2 N \rceil \right) \times N$$

where S_{I2i} is the secondary index size per argument position "i"

3.2. Primary Index File Size

An entry in the primary index file consists of two fields: A pointer to a given block in the secondary index file. and the BR of the first argument value in that block. The number of blocks in a secondary index file. denoted N_{bI2}. is equal to $\left\lceil \frac{S_{I2}}{B} \right\rceil$. where B is the block size in bits block Each block address is thus encoded in $\lceil \log_2 N_{bI2} \rceil$ bits. and the number of entries in the primary index file of a given argument "i" is equal to the number of blocks of its secondary index file so that the total size of the primary index file is:

$$S_{index2} = \sum_{i=1}^{N_r}((\lceil \log_2 \frac{N}{C_i} \rceil - \lceil \log_2 \lceil \frac{S_{I2i}}{B} \rceil \rceil) \times \lceil \frac{S_{I2i}}{B} \rceil)$$

4. Query Response Time of TIL

The derived equations are based on the following general assumptions on the hardware and system models

1. A given BR is equally likely to be specified in a query

2. The primary and secondary indices are stored in contiguous secondary storage blocks and ordered with respect to the BR values so that a block can be searched in log time.

3. Buffer sizes are sufficient to hold the retrieved blocks.

4. The disk controller(s) include a comparator which is used to perform on the fly comparison so that partial overlapping of the primary index blocks retrieval and search is achieved. This enables us disregard the disk rotational latency time for the retrieval of successive blocks from secondary storage. Furthermore, all blocks relevant to a given index file are assumed to reside on the same cylinder in secondary storage. This assumption holds as the index file sizes under consideration is relatively small.

5. Main processor comparison is word oriented and the time required to perform a comparison is $T_w \times \lceil \frac{BR}{WL} \rceil$ T_w being the average word comparison time and WL the word length of the main processor

6. We assume a stable file as defined by Larson [LAR81] and do not account, in our deterministic analysis, for the overhead incurred by searching overflow records according to Larson's stochastic model, the expected number of additional disk accesses required to search an indexed-sequential file is around 0.3 accesses

7. The hashing functions do not lead to collisions. However, in practice, collisions could be deleted by post checking of the retrieved records from the EDB prior to shipping them to the logic programming environment. This could be performed on the fly but is not included in the present analysis. Although not *required* for the analysis, *if order preserving hashing functions are provided* (Garg [GAR86]), TIL files could handle range queries as well

The Query Response Time (QT1) for TIL1 is divided into three processes:

1) Surrogate File Processing and Uid Retrieval (SFT1).

2) Uid Intersection Time (IT)

3) Database Access Time (DA) to read the identified record(s) satisfying the query.

It is noted that process 2 and 3 above are independent of the TIL type followed. The Query Response Time for the TIL1 technique is written

$QT1 = SFT1 + IT + DA$

The surrogate file processing time (SFT1) is subdivided into four sub-processes:

1 Primary Index Retrieval Time

2 Primary Index Search Time

3 Secondary Index Retrieval Time

4 Secondary Index Search Time

Due to assumption 4 above, primary and secondary index search times are neglected and we do not report them in this paper

4.1. Primary Index Retrieval Time

Fast sequential retrieval being followed the average number of blocks, N_{avgI1b}, retrieved for each query argument position is:

$$N_{avgI1b} = \frac{N_{I1b}+1}{2} = \frac{\lceil \frac{S_{I1i}}{B} \rceil + 1}{2}$$

With R_q as the number of arguments in a query, and sequential access of primary index blocks, the average TIL1 primary index retrieval time, $T_{index1-retrieval}$, is:

$$T_{index1-retrieval} = \sum_{i \in R_q} (T_{seek} + T_{lat} + \frac{(\lceil \frac{S_{I1i}}{B} \rceil + 1) \times B}{2 \times TR})$$

4.2. Secondary Index Retrieval Time

The TIL1 secondary index retrieval time is based on the average number of secondary index blocks, N_{avgib}, to be retrieved for each argument position in a query. The total secondary index retrieval time, $T_{index2-retrieval}$, is given by:

$$T_{index2-retrieval} = \sum_{i \in R_q} (T_{seek} + T_{lat} + \frac{N_{avgib} \times B}{TR})$$

and with C_{I2bi} as the number of entries in a secondary index block, N_{avgib} is computed in Appendix 1 as:

$$N_{avgib} = \frac{C_i}{C_{I2bi}} - \frac{1}{C_{I2bi}} + 1$$

4.3. Intersection Time

Two cases are considered:

$R_q = 1$: no intersection is required and the number of good drops (GD) is C_i.

$R_q > 1$ when more than one argument value is specified in a query, the lists of retrieved Uids must be intersected. Denoting by $NC(R_q)$, the number of comparisons required to perform the intersection operation, the total intersection time is then written as:

$$
IT = \begin{cases} T_w \times \left\lceil \dfrac{\lceil \log_2 N \rceil}{WL} \right\rceil \times NC(R_q) & \text{if } R_q > 1 \\ 0 & R_q = 1 \end{cases}
$$

An estimate of the number of comparison steps. $NC(R_q)$. for the intersection operation is derived in Appendix 2. As noted previously, we assume conjunctive queries with no loss of generality. as the union operation for disjunctive queries has the same level of complexity as the intersection operation

4.4. Database Access Time

With GD as the number of good responses to a query and the probability ($\frac{1}{\left\lceil \frac{S_{db}}{B} \right\rceil}$) of a given response to be in a specific block. the database access time is. following Cardenas' equation [CAR75] and assuming direct access to the main database:

$$
DA = (T_{seek} + T_{lat} + \frac{B}{TR}) \times \left\lceil \frac{S_{db}}{B} \right\rceil \times (1 - (1 - \frac{1}{\left\lceil \frac{S_{db}}{B} \right\rceil})^{GD})
$$

Following Appendix 2, the number of good responses is estimated as:

$$
GD = N \prod_{i \in R_q} (\frac{C_i}{N})
$$

It is observed that the database access equation is based on successive selection with replacement. Yao [YAO77] discusses selection without replacement and points out the cases where Cardenas' equation gives rise to a significant error For our purposes. Cardenas' approach is satisfactory as the number of good responses is expected to be small for very large knowledge bases.

5. Simulation and Analysis of TIL Techniques

In the following analysis and computer evaluation for the derived TIL equations, the parameters are as follows:

1) Each tuple in a database relation consists of a number of arguments, A_r, of 15 characters each.

2) In the derived equations. the C_i parameter is dependent on the argument position i To simplify those equations. we have used an average value for C_i, namely C_g.

3) The system variables for the simulation are given in Table 2.1 Disk parameters are based on the DEC RA81 hard disk system [KRI83].

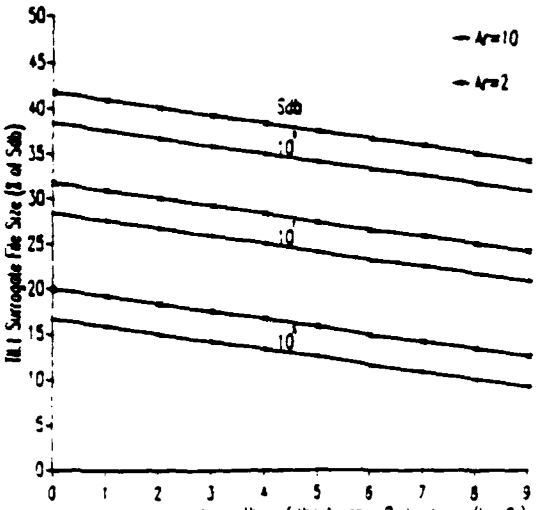


Figure 5.1 Effect of the Database Size and the Number of Arguments in a Tuple on the TIL1 Surrogate File Size.

Due to space limitations, all reported plots are based on a disk block size of 2 Kbytes.

In Figure 5.1, the TIL1 Surrogate File to Database size ratio is plotted versus the logarithm of the average redundancy factor, for different S_{db} and A_r values. In general the SF size of TIL1 spans from a low of 9.2%, for $\log_2 C_g = 9$, $A_r = 10$ and $S_{db} = 10^5$, to 41.8% for $\log_2 C_g = 0$, $A_r = 2$ and $S_{db} = 10^9$. It is noted that the plots in Figure 5.1 mainly reflect the variation of the secondary index file size as the primary index file size can be shown to be negligible. In [HAC87], the storage requirements for TIL2 are reported to range from 8 to 20% of the size of the database.

Figures 5.2 to 5.5 illustrate the TIL1 Query Response Time (QT1) and its corresponding subprocessing times (SFT1, IT and DA) for different database sizes and number of arguments in a query. Figures 5.2 and 5.3 relate to medium sized files ($S_{db} = 10^7$ bytes) while Figures 5.4 and 5.5 are typical of very large files ($S_{db} = 10^9$ bytes). It is observed that QT1 is highly dependent on the SF processing time (SFT1) for low values of C_g (up to 512) and then becomes highly dependent on the intersection time (IT) The drop in database access time (DA). observed between the plots of Figures 5.2 and 5.4 or 5.3 and 5.5, is due to the dependency of the number of good responses (GD) on the ratio $\frac{C_g}{N}$ For a fixed C_g, this ratio decreases with increasing database sizes.

No plots are included for the case where $R_q = 1$. In this situation, the query response time for TIL1 is dependent on the number of good responses which is C_g. Furthermore, TIL2 query response time variations are the same as for TIL1. The only difference being that TIL2 requires one additional disk access per query argument, that is balanced by a smaller disk transfer time for large values of the redundancy factor C_g. The disk transfer time is smaller due to a smaller surrogate file size.

We conclude that the TIL techniques are efficient as to the storage/query response time combination. Even for relatively large redundancy factors. the query response time is

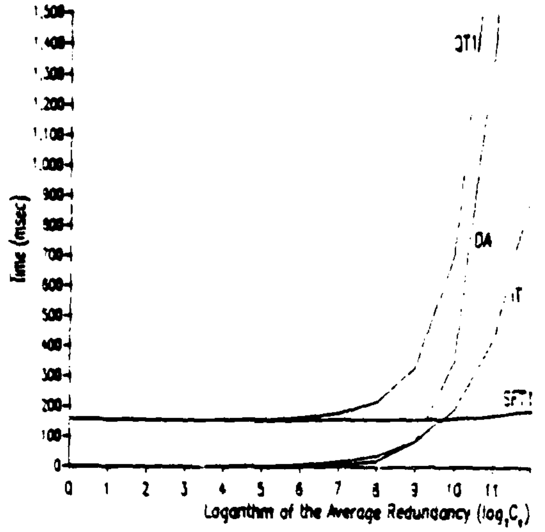


Figure 5.2 Components of the TIL1 Query Response Time (Sdb=10 bytes, Ar=6, Rq=2).

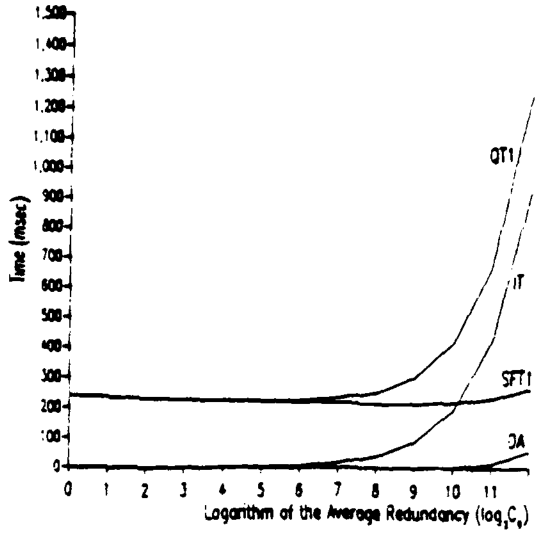


Figure 5.4 Components of the TIL1 Query Response Time (Sdb=10 bytes, Ar=6, Rq=2).

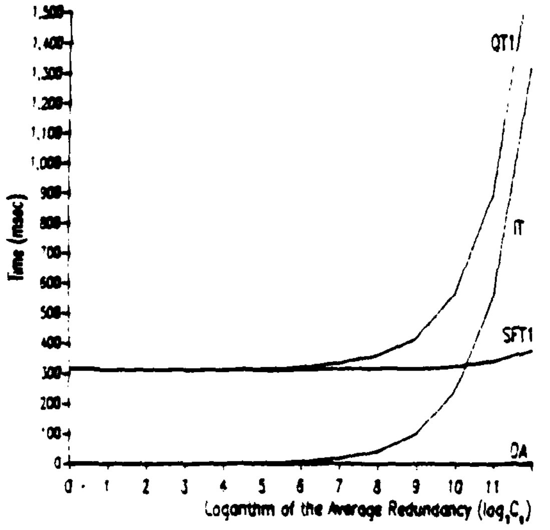


Figure 5.3 Components of the TIL1 Query Response Time (Sdb=10 bytes, Ar=6, Rq=4).

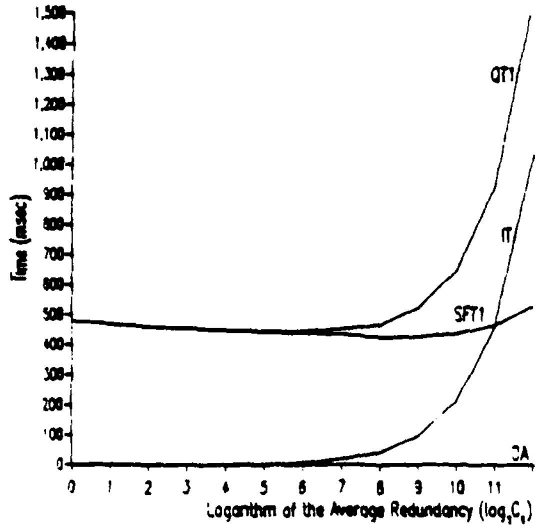


Figure 5.5 Components of the TIL1 Query Response Time (Sdb=10 bytes, Ar=6, Rq=4).

ithin a few seconds while the storage overhead of the surr-ate files lies in the 10 to 20 % range of the database size. It is noted that conventional inverted lists, with full indexing, may require an overhead well in excess of 100 % of the latabase size.

6. Maintenance Aspects of TIL Surrogate Files

One of the difficulties in using the TIL techniques is their maintenance requirements. Those become a serious drawback, especially in a highly volatile database environment. The above analysis pertains to a static surrogate file. If, for example, 30% expansion of the main database is forseen, the overall increase of the surrogate files sizes can be greate. than 30%, due to the additional increase required for the different record pointers and unique identifiers.

Some important maintenance aspects are the add, delete and update operations. When adding a new record to the database, all the index files have to be accessed and reordered, which is a time consuming operation. The use of overflow blocks would decrease the time requirements for the insert operation with a negative impact on query response time. Block inserts could be followed but this technique is not applicable to real time databases. In any case, periodical time consuming reordering is necessary. Deleting records could b performed by marking techniques and delaying reordering and packing operations to off line maintenance periods. Finally, updates require the access and rearrangement of the affected attribute's indices.

It can be stated, in general, that the overall management system requirements for TIL Surrogate Files is complex

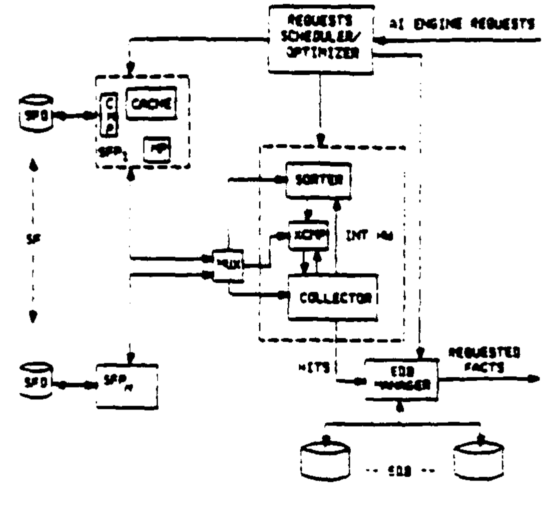


FIGURE 7.1 BACK END ARCHITECTURE FOR FACT MANAGEMENT.

and those techniques are not recommended in volatile database environments.

7. Back End Architecture for Knowledge Base Management

In this Section, we describe and analyze the benefits of a parallel back end architecture for the management of knowledge based systems with TIL surrogate Files.

7.1. Back End System

Shown in Figure 7.1 is a back end system for the management of a very large extensional database of facts. This system will also manage many intentional databases (sets of inference rules), but those are not shown on the diagram. We assume that there are many gigabytes of fact data stored on the EDB disks. Likewise, there are several gigabytes of surrogate file data stored on the SF disks (SFD). Since we have assumed the relational model we will store the facts by relation and then by tuple unique identifier within relations. As previously mentioned we will access the EDB only by relation name and then by tuple identifier, so extendible hashing or some such technique that minimizes disk accesses can be used

As an example, assume that a user's request requires access to only two lists. The relevant block(s) from the first list would be retrieved from the SFD and input to its associated surrogate file processor (SFP) where on the fly comparisons are made for matches by the comparator (CMP). Note that the SFP consists of a comparator (CMP) and cache (CACHE) with their associated control microprocessor (MP). The unique identifiers would be stripped off and sent to the Intersector Hardware block (INT HW) through the multiplexer (MUX). The list of Uids is piped in the pipeline sorter (SORTER) and then fed to the cross-lists comparator (XCMP).

Meanwhile, the second list is processed in a similar way and sent to the XCMP module. Then, the two resulting lists of possible responses are intersected by the XCMP block. The output of Uids (if any) is sent to the collector (COLLECTOR) that acts as a buffer and the block of good responses (HITS) is passed on to the Extensional Data Base Manager (EDBM) for processing. The EDBM will retrieve the facts, compare them with the search criteria to insure that a collision has not occurred, put them in blocks, and sends them to the logic programming engine.

In the case where more than two lists are to be intersected, the outcome of the two lists intersection is fed back from the COLLECTOR to the XCMP block for a new cross comparison operation with the third list coming from the SFD/SFP pairs. This process is continued until all the arguments in the query are properly processed. When a single argument query is considered, the MUX passes the incoming list from the SFD/SFP pair to the COLLECTOR that relays it to the EDB manager. The complete system can be viewed as a three level pipeline controlled by the Requests Scheduler-Optimizer.

7.2. Analysis of the Proposed Architecture

In this Section, we analyze the motivations and the benefits of the described architecture. One recurrent criticism against the use of inverted file structures is that their performance degrades as the number of arguments in a query increases. A good algorithm would tend to perform in the opposite way, as one hopes to do work proportional to the expected number of tuples in an answer. This criticism is assessed based on the sequential processing of the surrogate inverted lists, but is mitigated *if parallel processing algorithms* running on multi-processor architectures are designed for transformed inverted lists. We will have to look at the equations for the different subcomponents of the query response time for TIL, namely SFT1, IT and DA.

i Surrogate Files Processing Speedup

From the equations, derived in Sections 4.1 and 4.2, for the TIL1 surrogate files processing time (SFT1) and Figures 5.2 through 5.5, we observe that SFT1 is proportional to the number of arguments in a query (R_q) and is related to the disk access cost for the retrieval of the inverted lists indices. The TIL structure is well suited for parallel processing through the distribution of the inverted lists to multiple storage and associated processor units (SFP). For the case of a single user[21] queries on a relation with degree "d", an O(d) speedup for the surrogate files processing time can be achieved with a maximum of "d" SFD/SFP pairs. For a multi-user system, the speedup which can be achieved is a function of the number of SFD/SFP pairs and the application being considered. The surrogate file will actually consist of many sets of inverted subfiles, one set for each relation. Those sets will be distributed over the SF disks in order to insure maximum parallelism in disk accessing.

The distribution algorithm follows an optimization criterion related to the application on hand. We note that the

[21] A "user" is referred to as the application programmer. A single user refers to a single application environment versus a multi-user i.e multiple applications environment

assignment problem is NP_Complete and heuristic algorithms specifically designed for the proposed architecture. are being presently developed for the proper distribution of the surrogate inverted lists. The outcome of the algorithm would be a storage mapping of the surrogate inverted lists that is used by the Requests Scheduler Optimizer for query optimization.

Disk access cost can be further reduced by the use of cache memory in each SFP unit This cache would store the primary indices which are relatively small in size. With a cache hit ratio of 0.9. the average number of disk accesses. per inverted list. drops to 1.1 from the value of 2 that is assumed in the equations for SFT1 In practice. the number of disk accesses. per inverted list search. is slightly higher due to the overflow chains that are bound to exist and which were not accounted for in our analysis [LAR81]

2. Intersection Operation Speedup

The equation for the intersection operation cost (IT) was derived in Section 4.3. From this equation and the analysis of Appendix 2, IT can be easily shown to heavily depend on C_i. While negligible for small databases. IT becomes a computation bottleneck for medium and large data/knowledge bases with high average redundancy factors (C_i) (See Figures 5.2 to 5.5). It is noted that the plots represent computed intersection time for equal attribute selectivities: for example if $R_q = 2$. the same C_i is assumed for both arguments in the query If we follow the reasoning that the probability of both arguments in the query having high redundancy factors is low. then our plots are pessimistic and realistic values for IT would be noticeably smaller. This argument can be made for any database size.

Nevertheless. for a VLDKB. the plots in Figures 5.2 to 5.5 reflect an essential need for special intersection hardware. referred to as the Intersector In Figure 7.1. the Intersector is part of the INT HW block and consists of the pipeline sorter (SORTER) and cross-list comparator (XCMP) units. The sorter is essential. following our discussion in Appendix 2. and shall be optimized to handle large lists of Uids as they present the computation bottleneck of the intersection operation The XCMP block is used to cross compare the sorted list of Uids from the output of the SORTER with an incoming list of Uids from a SFP

With L_{min} as the minimum length of the lists involved in the intersection operation. an $O(L_{min})$ computation steps could be achieved with the Intersector Compared with an $O(L_{min} \times \log_2 L_{min})$ computation steps of the best sequential algorithm. the speedup achieved with the hardware Intersector would be $O(\log_2 L_{min})$.

For high query rates. the operation of the INT HW block and the SFD/SFPs are overlapped. thus increasing the throughput of the system. The number of Intersector blocks is not bound to one. as shown in Figure 7.1. and is a function of the throughput constraint of the design. Maximizing the level of pipelining between the SFD/SFP pairs and the INT HW block(s) is an additional requirement on the optimization algorithm It is worth noting that a different intersection hardware could be derived based on a parallel cartesian product algorithm. We believe that such hardware would be more elaborate than the sorter/cross comparator combination.

3. Comments on the Database Access Time

Database access time (DA) depends on the locality of the good responses and would be determined by the clustering scheme for the tuples in the existing EDB In our analysis. DA is derived following Cardenas' assumptions [CAR75] of uniform distribution for the records over the EDB secondary storage blocks. In a multi-user environment. clustering can achieve optimal DA values for one user while degrading the response time for another EDB clustering is an open design problem that lies in the class of NP_Complete problems. Its discussion is not within the scope of this paper

8. Conclusion

In this paper. we presented the equations to estimate the storage overhead and query response time for Transformed Inverted Lists. Surrogate files based on TIL were found to be efficient as to a space/time criteria. While the size of the TIL files is larger than the ones for other techniques like Superimposed and Concatenated Code Words [BER87], it lies within an acceptable range of storage overhead (10 to 30 % of the database size). The superior partial match response time of the TIL file structures is an asset for their use in the context of a Very Large Knowledge Base.

TIL surrogate files are found to be well suited for parallel processing with multiple storage/processor units. Based on TIL file structures, we described and presented a preliminary analysis of a parallel back end architecture for partial match queries on a VLDKB. Our current research is directed towards the development of the proposed back end system. based on current and additional results. Many more issues shall be addressed such as updating, integrity, collisions and adapting the inverted surrogate files to volatile data/knowledge bases. Another open research problem we are studying is the development of optimal allocation algorithms for the surrogate inverted lists on multiple storage/processor units.

Acknowledgement

This work was supported by the Air Force Systems Command. Rome Air Development Center. Griffiss Air Base. New York 13441-5700. and the Air Force Office of Scientific Research. Bolling AFB. DC 20332 under Contract No. F30602-85-C-0008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC).

Appendix 1: Average Number of Adjacent Blocks Containing the Same Argument Value

The average number of consecutive blocks containing records with the same i_th argument value in an index file is derived The following terms are defined:

1) C_i. the number of records with the same i_th argument value.

2) C_b. the number of records in an index file block.

3) X: number of consecutive blocks as a random variable. We have to compute E(X), the expected value of X.

It is noted that the C_i records reside consecutively in an index file, and the first record can be located at any k_th position in an index block with equal probability. Three cases are considered.

1) $C_i < C_b$. The number of blocks to be retrieved is either 1 or 2 blocks. We can write.

$$P(X = 2) = \frac{C_i - 1}{C_b} \text{ and } P(X = 1) = 1 - \frac{C_i - 1}{C_b}$$

2) $C_i > C_b$, with $C_i = n \times C_b + r$ and $r \neq 0$:

$$P(X = \left\lceil \frac{C_i}{C_b} \right\rceil + 1) = \frac{(C_i \mathrm{mod} C_b) - 1}{C_b} \quad \text{and}$$

$$P(X = \left\lceil \frac{C_i}{C_b} \right\rceil) = 1 - \frac{(C_i \mathrm{mod} C_b) - 1}{C_b}$$

$$\text{with } C_i \mathrm{mod} C_b = C_i - \left\lfloor \frac{C_i}{C_b} \right\rfloor \times C_b$$

3) $C_i = n \times C_b$. In this case we write.

$$P(X = \frac{C_i}{C_b} + 1) = \frac{C_b - 1}{C_b} \text{ and}$$

$$P(X = \frac{C_i}{C_b}) = \frac{1}{C_b}$$

For all three cases, it is easily shown that the expected value of X, E(X), is governed by the following equation:

$$E(X) = \frac{C_i}{C_b} - \frac{1}{C_b} + 1$$

Appendix 2: Estimating the Number of Required Comparisons for the Intersection Operation

Stockmeyer and Wong [STO79] give the following bounds on the number of comparisons, I(m,n,k), required to intersect two lists, m and n, of arity k ($m \leq n$).

$$I(m,n,k) \leq (m + n) \times \log_2 m + (m + n - 1) \times k - m + 1$$

$$I(m,n,k) \geq \mathrm{Max}\,(m + n) \times \log_2 m - 2.9m, (m + n - 1) \times k - m + 1$$

In our case the arity k=1 and number of comparisons, NC(2), to intersect two lists of cardinalities $C_1 \leq C_2$ is

$$NC(2) \geq \mathrm{Max}((C_1 + C_2) \times \log_2 C_1 - 2.9C_1, C_2,$$

$$NC(2) \leq (C_1 + C_2) \times \log_2 C_1 + C_2$$

The upper bound is based on sorting the list of smaller cardinality prior to performing the cross lists comparison in at most $C_2 \times \lceil \log_2 (C_1 + 1) \rceil$ comparisons. It is known that two_way merge sort on a uniprocessor requires at most $C_1 \times \log_2 C_1$ comparison steps. It is easy to derive an algorithm that would perform within the specified bounds [KNU73]. Furthermore, if we need to intersect more than 2 lists, the number of additional comparisons depends on the expected number of "hits" from the first two_list intersection. Denoting this number by GD, $GD = \frac{C_1 \times C_2}{N}$, where N is the number of records in the database. For $R_q = 3$, we need $C_3 \times \log_2 \lceil (GD + 1) \rceil$ additional comparisons. So that NC(3) is written as:

$$NC(3) \leq NC(2) + C_3 \times \log_2 \left\lceil \frac{C_1 \times C_2}{N} + 1) \right\rceil$$

The process can be extended to include additional intersection steps for larger values of R_q. It is noted that Cardenas [CAR75] does not attempt to give an estimate of the intersection time and Frederowicz's approach [FRE87] is different than ours.

As to the number of good responses (GD), we wrote, for $R_q = 2$:

$GD = \frac{C_1 \times C_2}{N}$ Assuming uniform distributions for the values of an argument, the number of good drops can be extrapolated to:

$$GD = N \prod_{i \in R_q} (\frac{C_i}{N})$$

References

[AHU80] Ahuja S.R., Roberts C.S., "An Associative/Parallel Processor for Partial Match Retrieval Using Superimposed Codes," Proc. 7th Annual Symp. on Computer Architecture, August 1980, pp 218-227.

[BER87] Berra P.B., Chung S.M., Hachem N.I. "Computer Architecture for a Surrogate File to a Very Large Data/Knowledge Base" IEEE Computer, March 1987, pp 25-32.

[CAR75] Cardenas A.F. "Analysis and Performance of Inverted Data Base Structures," Communications of the ACM Vol. 18, No. 5, May 1975, pp 253-263.

[COL86] Colomb R.M. Jayasooriah," A Clause Indexing System for Prolog based on Superimposed Coding," The Australian Computer Journal, Vol. 18, No. 1, February 1986, pp 18-25.

[DAT86] Date C.J "An Introduction to Database Systems, Volume I" Addison-Wesley Systems Programming Series, 1986.

[FAG79] Fagin R., Nievergelt J., Pippenger N. and Strong H.R. "Extendible Hashing-A Fast Access Method for Dynamic Files." ACM Transactions on Database Systems, Vol. 4, No. 3, September 1979, pp 315-344.

FAL85 Faloutsos C. "Access Methods for Text." Computing Surveys. Vol.17. No. 1. March 1985. pp 49-74

FRE87 Frederowicz J. "Database Performance Evaluation in an Indexed File Environment" ACM Transactions on Database Systems. Vol. 12. No. 1. March 1987. pp 85-110

GAR86 Garg A.K. Gotlieb C.C. "Order-Preserving Key Transformations." ACM Transactions on Database Systems. Vol. 11. No. 2. June 1986. pp 213-234.

HAC87] Hachem N.I. Berra P.B. "Parallel Architecture for Transformed Inverted Lists. A Surrogate File Structure for a Very Large Data/Knowledge Base" ECE Department Technical Report. Syracuse University. June 1987

KNU73. Knuth D.E. "The Art of Computer Programming. Sorting and Searching". Volume 3. Addison-Wesley Publishing Co. 1973

KRI83] Kriddle B. and McKusick M.K. "Performance Effects of Disk Subsystem Choices for VAX systems Running 4.2 BSD UNIX." U.C. Berkeley C.S Department Technical report . 1983.

LAR81] Larson P. "Analysis of Index-Sequential Files with Overflow Chaining" ACM Transactions on Database Systems. Vol. 6. No. 4. December 1981. pp 671-680.

LEE86] Lee D.L. "A Word-Parallel. Bit-Serial Signature Processor for Superimposed Coding," International Conference on Data Engineering, IEEE-CS. February 1986. pp 352-359

LLO80 Lloyd J.W. "Optimal Partial-Match Retrieval," BIT 20. 1980. pp 406-413

LLO82. Lloyd J.W. and Ramamohanarao K. "Partial-Match Retrieval for Dynamic Files." BIT 22. 1982. pp 150-168.

ROB79] Roberts C.S. "Partial Match Retrieval via the Method of Superimposed Codes." Proceedings of the IEEE. Vol 67. No. 12. December 1979. pp 1624-1642.

STO79] Stockmeyer L.J. Wong C.K. "On the Number of Comparisons to Find the Intersection of Two Relations" SIAM Journal on Computing, Vol. 8. Nb. 3. August 79. pp 388-404.

WIS84] Wise M.J. and Powers D. "Indexing Prolog Clauses via Superimposed Code Words and Field Encoded Words." International Symp. on Logic Programming. February 1984. pp 203-210.

YAO77] Yao S.B. "Approximating Block Access in Database Organization" Communications of the ACM. Vol. 20. No. 4. April 77. pp 260-261.

Appendix 9-C

AN ARCHITECTURE FOR VERY LARGE RULE BASES BASED ON SURROGATE FILES[1]

DONGHOON SHIN
P. BRUCE BERRA

Syracuse University, Syracuse, New York 13244-1240, USA

ABSTRACT

To support a large set of rule bases as well as ground facts, we propose an efficient retrieval method by transforming heads of clauses and facts into Concatenated Code Words (CCW) to form a surrogate file. By adopting the 'mode' declarations used in PARLOG, the heads of clauses can be represented by function-free terms, and then are transformed to CCW to be used as an index to gain access to the actual database. A simplified unification operation on surrogate files can be efficiently implemented by means of a specialized associative processor due to the uniform structure of surrogate files.

[1] This work was supported by the Air Force Systems Command, Rome Air Development Center at Griffiss Air Force Base in New York, and the Air Force Office of Scientific Research at Bolling AFB in Washington DC under Contract No. F30602-85-C-0008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC).

INTRODUCTION

Future computer systems will be expected to provide highly efficient management of large shared knowledge bases for knowledge-directed applications such as expert systems. Previous knowledge base systems such as ILEX (1) and DELTA (2) have the dual structure consisting of an inference engine and a knowledge base. These have attempted to combine a relational database system to manage the knowledge base with a logic programming system to serve as the inference engine. For efficient management of a large database, the Extensional Database (EDB) is separated from Intensional Database (IDB). Though this approach has exhibited a great deal of efficiency for handling a large set of facts (EDB), it may not be suited to applications supporting large rule bases (IDB) which heretofore have been assumed to be small enough to reside in the main memory. It has also been observed that most inefficiencies stem from the interface between these two very different systems.

On the other hand, in some recently proposed systems, there is no distinction between the IDB and EDB. That is, both facts and rules are managed and stored uniformly. A machine that uses the idea of database retrieval based on the unification operation is the Sabbatel's Prolog database machine (3). It can search desired data form secondary storages by the "on the fly" execution of unification. Sabbatel proposed the Prolog's top-down evaluation strategy with AND/OR parallelism and set-oriented processing to reduce the number of accesses to secondary storage. Recently, Yokota and Itoh proposed the "Relational Knowledge Base Model" to provide a machine with a uniform representation of the knowledge base (4). Unlike the relational database model that consists of only ground instances, this model can accommodate variables and complex structured terms. In this case, the exact match of database operations should be extended to unification due to the variables and structured terms that can appear in the knowledge base. However, the processing load required for such an operation on a large knowledge base stored in secondary storage is expected to be enormous. Furthermore, this approach can be inefficient because of the 'top-down' query evaluation strategy, especially when a large set of ground facts are involved.

Presented in this paper are techniques for managing a very large knowledge base to support diverse requirements for applications of logic programming systems based on surrogate files (5) and associative processors. We also propose an integrated knowledge base machine architecture that can effectively support very large sets of rules as well as facts in the context of logic programming environment.

This paper consists of 6 sections. In the next section we give some basic definitions followed by restricted representations of clause heads to be used to form a surrogate file. In section 3, we present the basic method of constructing a surrogate file for rules and facts. Section 4 describes the basic idea for unification on a surrogate file and an associative processor to realize it. Section 5 presents the architecture of the proposed knowledge base machine and its parallel processing model. Finally, in section 6, we present some conclusions and suggestions for future work.

PRELIMINARIES

Conery (6) has classified the inherent parallelism in logic programming systems into three major categories: AND-Parallelism, OR-Parallelism and Low-level Parallelism. Our major concern here is a special case of OR-parallelism called search parallelism which has been defined as a parallel distributed search to find every clause with a head that unifies with the selected goal. Since a search performed by integrated knowledge base machines should be based on unification rather than equality, it is well known that an efficient implementation of unification is the central issue in logic based systems. Several processors dedicated to the unification operation have been proposed in recent years to accelerate this most time-consuming operation in logic programming evaluation (7)(8) (9).

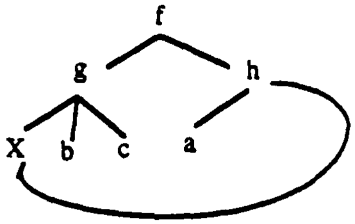
Informally, the main purpose of unification is to make two or more terms identical by proper and the most general substitutions for logical variables in the terms. A term is defined as follows (10):

(1) A variable is a term denoted by a capital letter such as X,Y,Z,...

(2) A constant is a term denoted by a lower case letter such as a,b,..

(3) If f is an n-ary function and $t_1,...,t_n$ are terms, then $f(t_1,...,t_n)$ is a term.

Ever since Robinson introduced the basic algorithm of the unification operation for the resolution principle (11), more efficient algorithms have been proposed and the complexity of the unification operation has been analyzed by many researchers (12)(13). Among them, two algorithms (14)(15) are claimed to be linear. These algorithms are based on a complex data structure called Directed Acyclic Graph (DAG). Also, Morita proposed a linear representation of a term suited to stream processing of unification (16). The DAG and linear representations of a term are shown in Fig. 1 (a) and (b) respectively.



(a) DAG

(f2)(g3)(X0)(b0)(c0)(h2)(a0)(X0)

(b) Charcter String

Fig.1 The Representations of a Ter m(f(g(X,b,c), h(a,X))

Our major concern in implementing unification for very large rule bases in secondary storage, is finding all potential candidate clauses within a small amount of time so that we can deal with real time applications. Since the full unification on such data will require a heavy processing load, our goal may not be achieved without restricting unification. Furthermore, the results of (12) indicate that, since unification is inherently sequential, even parallel evaluation of a unification algorithm may not offer a considerable speed-up over a sequential one.

The major processing load stems from 'occur checks' to prevent the unification from entering an infinite loop. That is, when testing if a variable X unifies with a structured term t, a check should be done whether X occurs in t (i.e. {X/f(X)}). We can eliminate these requirements by adopting mode declarations to construct a 'standard form' of clauses as in PARLOG (17) where the structured arguments appearing in clause heads can be transferred to the bodies of clauses.

A PARLOG program that possesses a single solution consists of a sequence of guarded Horn clauses. A guarded Horn clause of PARLOG has the form

$A\text{:-}G_1,G_2,..G_m:B_1,B_2,...,B_n.$

$m,n\geq 0$

If m=0 then the commit oprerator can be omitted. A candidate clause of PARLOG is one which succeeds in all input matching with the call (subquery) and whose guard literals ($G_1,G_2,..,G_m$) are proven to be true.

PARLOG exploits "mode" declarations for the clauses in the single solution relation to avoid the requirement of full unification, and to control process synchronization (17). A mode declaration for a predicate can constrain the unification between a goal and a clause (head) in a program. Mode declaration is of the form

mode $R(m_1,m_2,....,m_k)$

where R is a predicate name and each m_i is either '?' or '^'.

An argument annotated with a '?' in the mode declaration for a predicate can only be used for input matching against the corresponding argument of a call. That is, the unification between a call and the head of the clause is successful only if the corresponding argument in the call is instantiated (i.e. not a variable). Otherwise the evaluation suspends. On the other hand, an argument annotated with a '^' must be used for output matching against a variable of the corresponding position of a call. In other words, the corresponding argument of a call should be an uninstantiated variable on unification. If the argument is not an uninstantiated variable, the unification fails.

The mode declaration is used to determine the 'standard form' of clauses at the first stage of compilation. In the standard form, all complex terms appearing in the heads of clauses can be represented as pure variables, and all input and output matching between a call and the heads of clauses are translated to explicit unification primitives instead of general unification.

Consider, for example, a simple PARLOG program

```
mode member(?,?).
member( H,[H|T] ).
member( H,[X|T] ) :- ~H=X : member(H,T).
```

where ':' is the commit operator and ~H=X is a guard.

This program can be mapped into the standard form

```
member(H,Y) :- [X|T]<=Y,H=X:.
member(H,Y) :-[X|T]<=Y,~H=X: member(H,T).
```

The term [X|T] that was in the second argument position of the second clause head appears as [X|T]<=Y because it has the mode '?'. Here '<=' is the one way unification primitive that can only bind variables in its left argument([X|T]). This implies that this term can only be used for input matching against the given argument Y of the call. The repeated use of the term H in the head of the first clause is detected as an implicit test because both terms have the mode '?'. Thus the term [H|T] is changed to [X|T] (here X is an arbitrary variable) and an explicit test unification primitive '=' is added in the guard. In order to change a non-variable term with the mode '^' to the standard form, the assignment unification primitive ':=' should be used in the body. The unification primitives of PARLOG are described in (17). Maluszynski and Komorowski (18) have also discussed the use of mode to constrain full unification.

Consequently, the structured arguments (e.g. [H|T]) in the clause head can be transferred to the guard or body of a clause as shown in the above examples.

SURROGATE FILES

Surrogate files are constructed by hashing transformation of terms. The principal techniques that we have considered for the construction of the surrogate file include concatenated code words (CCW), superimposed code words (SCW), combinations of CCW and SCW, and transformed inverted list (TIL) (5). But, we will use only CCW to illustrate the ideas.

Suppose we have a fact called parent(timothy,johnson). We would first hash the individual values of each argument,

H(timothy)   H(johnson)
    | |
010111111   010110000

concatenate them, and then attach a unique identifier to obtain the CCW

010111111 | 010110000 | uid

where the vertical line shows the boundaries.

The same unique identifier would also be added to the actual fact itself so that a CCW can be used as an entry for each fact via the unique identifier.

This technique has been used for partial match retrieval on large set of facts with varying degrees and cardinalities. In retrieving facts, we assume that the facts are stored in such a way that one first accesses the relation and then a particular tuple using a unique identifier. Thus, we do not need to transform the predicate name (e.g. parent) for the facts. We obtain the unique identifier from processing the surrogate file, and the name of the relation from the given query. Thus, the storage structure for the facts themselves would be very simple and the desired facts can be retrieved in at most two disk accesses.

Most relational operations such as selection and join, which are required for the bottom-up query processing in logic-oriented database systems, can be performed on the surrogate file rather than on the actual database. This makes relational operations much faster and increases the system's performance when a large volume of ground facts exist.

In a CCW representation of a clause head containing variables, we do not consider structured terms and assume that the clause head contains pure variables and constants as arguments based on the transformation technique by adopting the mode declaration.

Variables should be distinguished from constants. This can be done by setting the msb (most significant bit) of the CCW to ' 1 '. Unlike facts, there are only a small number of rules that define a predicate, i.e. rules with the same head. Thus, we need to transform the predicate name as well as arguments.

Suppose we have rules for 'ancestor',

ancestor(X,Y):-parent(X,Z),ancestor(Z,Y).

ancestor(X,Y):-parent(X,Y).

We hash the predicate name and arguments by the same hashing function used in CCW for facts. The number of arguments is also concatenated to the hashed value of predicate name.

H(ancestor)	2 (No. of Arg.)	H(X)	H(Y)
|	|	|	|
011100000	0010	100100111	100101001.

The CCW representations for the two rules would be the same except for the uid's to be attached to them.

0111000000010 | 100100111 | 100101001 | uid_1
0111000000010 | 100100111 | 100101001 | uid_2

Thus, a surrogate file can be used to find the corresponding bodies of clauses with which a goal can unify via uid's.

This method guarantees retrieval of all desired terms (clause heads or facts) although, due to possible collisions resulting from the hashing method some undesired terms may be retrieved. A longer word length for the CCW can minimize such collisions, and post retrieval comparisons can be used to eliminate unwanted terms.

In the next section, we describe how one might perform unification on a surrogate file by proposing a special associative memory for bidirectional don't care matches.

UNIFICATION ON SURROGATE FILES

In this section, we present the basic idea of unification on a surrogate file using an associative processor. We have shown in section 2 how to transfer the complex structured arguments in the head of a clause to its body. For simplicity, we assume that the query contains only pure variables and constants. Thus, the Query Code Word (QCW) can be encoded by the same technique as described in section 3.

First, for all constants in a QCW, the corresponding arguments of the CCW must be either the same constant or a variable in order for the terms to be unifiable (Input matching Condition).

In the input matching step, we regard all variables as "don't care match" indicators. Unlike usual "don't care" matches, however, we need bidirectional don't care matches because the data residing in associative memory, as well as the QCW, may also contain variables. Since general associative memories do not provide this capability, a special associative memory is required. We designed an enhanced associative memory for bidirectional don't care matches, as shown in Fig. 2. Since by assumption only variables and constants appear in a QCW, input matching among a QCW and a number of CCW's, each representing a head of a clause, can be performed in O(1) time (i.e. constant time).

By input matching, most unqualified terms can be pruned. After input matching, we assume that the qualified terms (heads) are read one by one for further processing. Thus post processing will be required for only a relatively small number of terms, namely the qualified terms.

Obviously, the above condition is not sufficient. Consider, for example, two terms of the form q(a,X,b) and q(Y,a,Y). Though they satisfy the condition, they are not unifiable. We need post processing for the shared variables that appear in arguments of qualified CCW's. If the same variable appears in arguments of a CCW, they should be bound to the same constant or variable (Input matching consistency).

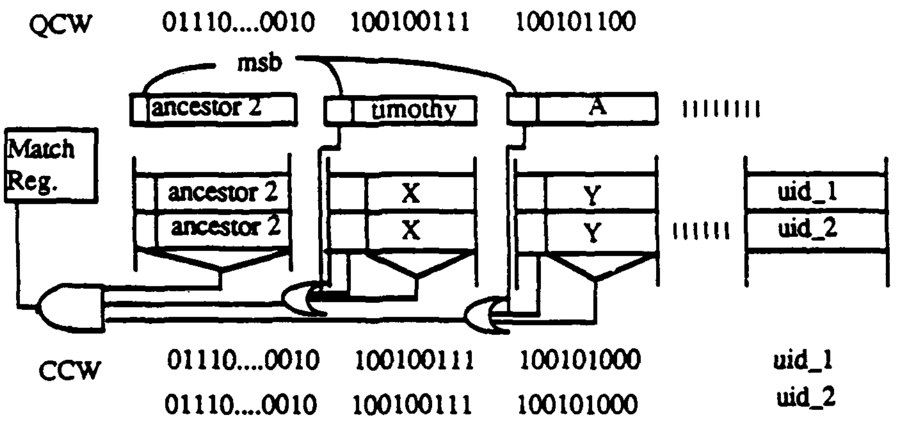


Fig. 2 An Associative Memory for CCW

The prime objective of unification is to find proper bindings for variables. After input matching and consistency checking are performed, the variables of qualified terms (CCW's) are substituted by the constants obtained from input matching. The reverse operation is required to bind variables in QCW. If these terms are unifiable, then the similar condition as the input matching condition will be satisfied. That is, for all constants in a qualified CCW, the corresponding arguments of QCW should be either the same constants or variables (Output matching condition).

Finally, a consistency check for the variables in the QCW needs to be performed. That is, if the same variables appear in the arguments of the QCW, they should be bound to the same constant or variable (Output matching consistency).

The unification method always works with the function-free terms. In the next section, the overall architecture and a processing model, as an example of parallel evaluation of logic programs, are described.

THE KNOWLEDGE BASE MACHINE ARCHITECTURE

The knowledge base machine architecture for surrogate file processing consists of four major components (Fig. 3):

1) A control processing element

(Control Processor(CP) + Main Memory),

2) A database manager,

3) A high speed shared memory and

4) Several surrogate file processors (SFPs).

The Control Processor can be a general purpose high performance processor. The main memory can be viewed as a local memory of the CP. In the logic programming framework, the CP performs the resolution (variable substitution) and accesses the actual KB. In our logic programming framework, we assume that the clause heads and facts are stored across distributed surrogate files under SFPs. The clause bodies, on the other hand, are contained in the database which is controlled by the control processor.

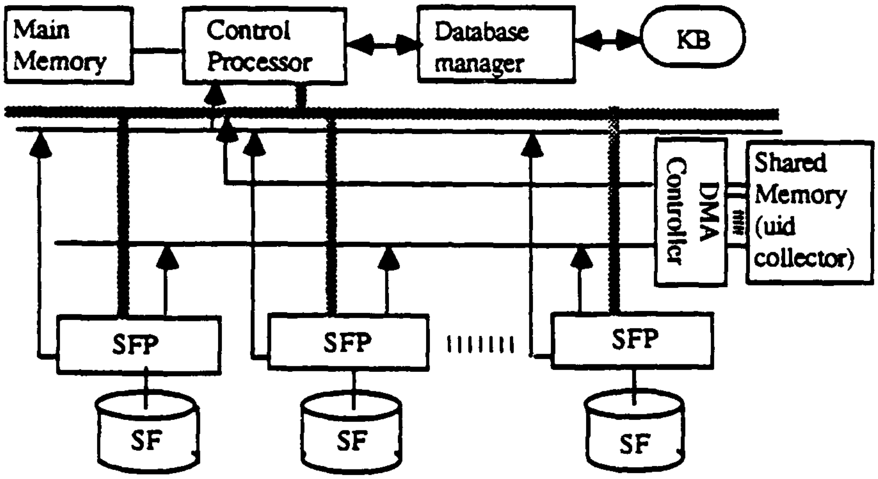


Fig. 3 Proposed Knowledge Base Machine Architecture

Our system can be viewed as a shared-memory system which is a tightly coupled multiprocessor that provide all SFPs equal access privileges to the shared common memory. Because of the tight coupling between processors and memories, this system can exhibit high performance. As can be seen in Fig. 3, SFPs do not need to communicate with each other. That is, the unification operation is local to each SFP, the CP does not access the local memories of the unification processors, and a SFP is not allowed to access the main memory. All the communications required between the CP and SFPs are performed by accessing shared memory. The contents of shared memory, once written by a SFP as a result of a successful unification, are not changed until a new initial goal is to be executed. Since the data in the shared memory is always valid, whenever the shared memory gets new data from a surrogate file processor, the CP can read the data. The maximum performance is achieved when the CP does not have any idle time.

As shown in Fig. 4, to prevent possible contention problems, we propose to use high speed shared memory and to give the CP a higher priority in accessing (read) the memory than the SFPs (write).

Since our architecture incorporates several SFPs for unification, OR-parallelism can effectively be exploited in top-down evaluation of a query. AND-parallelism, however, may not give us a considerable speed-up due to the binding conflicts among shared variables. Consequently, an OR-parallel/AND-sequential processing model with breadth-first search strategy is currently considered. Due to its breadth first search nature, the resulting model is in some respects similar to the LPS algorithm of DADO (19).

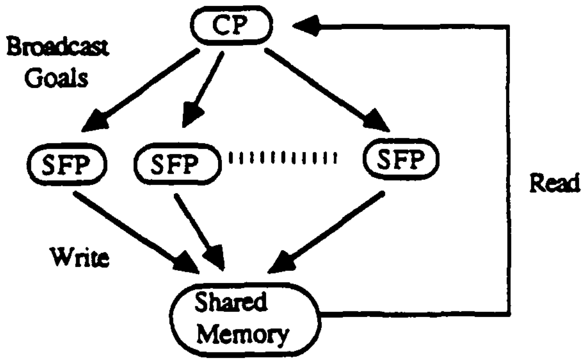


Fig. 4 The Sequence of Data Paths in Run Time

The CP broadcasts the initial goal to each SFP, where the surrogate file is managed and unification is performed. A processor that succeeds in a unification, accesses the shared memory to write the variable bindings and uid. The uid can be used by the CP to identify the corresponding body portion of a qualified head. The control processor resolves the body literals with the bind'ngs and broadcasts the subgoals one at a time. The flow chart of this method is presented in Fig. 5.

For example, to evaluate the goal :-? ancestor(timothy,X), the control processor broadcasts it to each SFP. Each SFP tests to see if an ancestor(timothy,A) can be unified with any header it contains by transforming the goal to a QCW. There will be two matches in our example, the one from ancestor(X_1,Y_1):-parent(X_1,Y_1) and another from ancestor(X_2,Y_2):-parent(X_2,Z_2), ancestor(Z_2,Y_2).

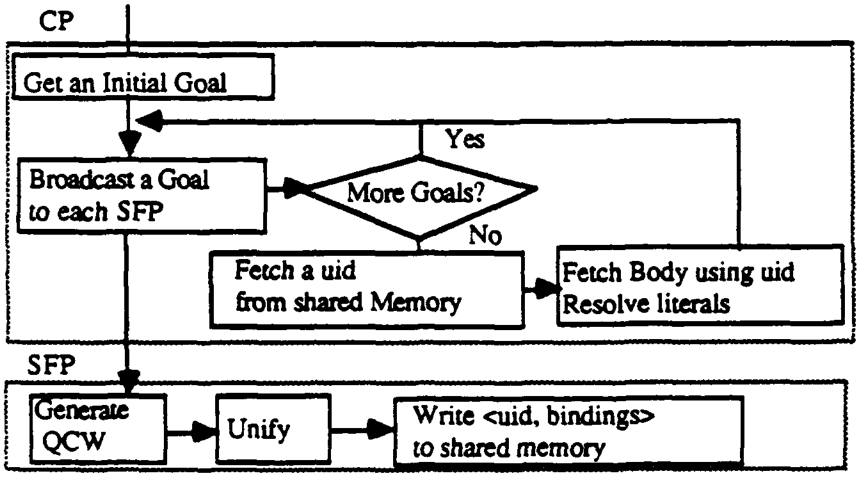

CP
Get an Initial Goal
Yes
Broadcast a Goal to each SFP
More Goals?
No
Fetch a uid from shared Memory
Fetch Body using uid Resolve literals
SFP
Generate QCW
Unify
Write <uid, bindings> to shared memory

Fig. 5 Logic Programming Evaluation based on Surrogate Files

Assume that the uid of the first clause is 'uid_1' and the one corresponding to the second clause is 'uid_2'. The control processor reads the shared memory to get the corresponding uids and variable bindings resulting from a successful unification. In our example, the contents of shared memory that can be accessed by the CP after broadcasting the initial goal would be either <{X_1/timothy, Y_1/A}, uid_1> or <{X_2/timothy, Y_2/A}, uid_2>. We do not care which clause succeeded first. A portion of the body corresponding to either uid_1 (i.e. parent(X_1,Y_1)) or uid_2 (i.e. parent(X_2,Z_2), ancestor(Z_2,Y_2)) is accessed from actual database via uid's, and the corresponding body is resolved by the bindings. That is, the variables which appeared in the body portion are substituted by obtaining them from shared memory. If the second clause is unified before the first one, the CP creates two AND processes of parent(timothy,Z_2) and ancestor(Z_2,A). Then the goal, parent(timothy,Z_2), is broadcast first.

In our processing model, if the SFP is efficient enough to make the control processor busy, the time required for unification is negligible. Hence a considerable amount of speed up can be gained in accessing secondary storages. The overall architecture is designed to exploit the advantages of both shared and private memory systems based on the top level algorithm described in Fig. 5.

CONCLUSION AND FUTURE WORK

We described surrogate file structures and a processing method that one might use to evaluate goals in top-down fashion when a large number of rules exist. When a large volume of facts are involved, the top-down query processing may be inefficient (20). In this case, a set-oriented, bottom-up query processing is more desirable than the top-down, tuple based one. Since the surrogate file technique has been originally designed for ground instances of facts, they can be effectively used for the bottom-up, set-oriented query processing in the framework of logic-oriented database systems. In addition, by separating the bodies (the actual codes for operations) from heads (an entry point for the procedure call), the surrogate file processing technique could support multiple knowledge representation schemes as well as conventional procedure- based, compiled languages.

We are currently approaching an efficient implementation of a knowledge base system in two ways. The first is to develop special hardware to process surrogate files; these files can allow efficient access to the knowledge base residing in secondary storages. The second is to consider optical techniques that can potentially increase data rates by orders of magnitude and thus speed access to the knowledge bases. This paper presented one of the first approaches.

REFERENCES

1. Li, D. A Prolog Database System, Research Studies Press, London,1984.
2. Murakami, K. et al. IEEE Computer, pp. 76- 92, June 1985
3. Sabbatel, G. B. et al. Proc. of the Second Int. Logic Programming Conference, pp. 207-217, 1984
4. Yokota, H. and Itoh, H. Proc. of the 13th Int. Symp. on Computer Architectures, pp. 2-9, 1986
5. Berra, P. B. , Chung, S. M. and Hachem, N. I. IEEE Computer, pp. 25-32, March 1987

6. Conery, J. S. , Parallel Execution of Logic programs, Kluwer Academic Publishers, Boston, 1987
7. Woo, N. S. Micro 18 Proceedings, pp. 89-98, 1985
8. Shobatake, Y. and Aiso H. Proc. of 13th Int. Symp. on Computer Architectures, pp.140-148, 1986
9. Stormon, C. D. CASE Center TR 8611, Syracuse University, October 1986
10. Lloyd, J. W. Foundations of Logic Programming, Springer-Verlag, 1984
11. Robinson, J. A. J. of the ACM, Vol.12, pp. 23-44 , 1965
12. Dwork, C., Kanellakis, P. and Mitchell, J. Journal of Logic Programming, Vol. 1, pp. 35-50, 1984
13. Vitter, J. S. and Simons, R. A. IEEE Transactions on Computers, Vol. C-35, No. 5, pp. 403 -418, 1986
14. Paterson M. S. and Wegman, M. N. Journal of Computer and System Sciences 16, pp. 158-167, 1978
15. Martelli, A. and Montanari, U. ACM Transactions on Programming Languages and Systems, Vol.4, No.2, pp. 258-282, April 1982
16. Morita, Y., Yokota, H. and Itoh, H. 12th VLDB, pp. 52-59, August 1986
17. Clark, K. and Gregory, S. ACM Transactions on Programming Languages and Systems, Vol.8, No.1, pp. 1-49, January 1986
18. Maluszynski, J. and Komorowski, H. J. Proc. of Second Int. Symp. on Logic Programming, 1985
19. Lowry, A., Taylor, S. and Stolfo, S. Proc. of the Int. Conf. on Fifth Generation Computer Systems, pp. 436-448, 1984
20. Kifer, M. and Lozinskii, E. Proc. of Third Data Engineering, pp. 375-385, 1987

Appendix 9-D

Optical Techniques and Data / Knowledge Base Machines

P. Bruce Berra

Nikos B. Troullinos

Dept. of Electrical and Computer Engineering

Syracuse University, Syracuse, NY 13244-1240

{berra,nicktrou}@sutcase.case.syr.edu

July 1987

This work was supported by 1) the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under Contract No. F306002-85-C-0008 which supports the Northeast Artificial Intelligence Consortium (NAIC) and 2) the New York State Science and Technology Foundation through Grant No. SSF (86)-26.

CONTENTS

Short statement

Optical storage, communication and processing hold the potential for two orders of magnitude performance improvement in data / knowledge base processing.

Introduction

The task of collecting, accessing and maintaining data, in all its forms, is the main concern of database management. Over the years it has been established as one of the most vital computer applications. With the advances of technology and the ever increasing dependency on computers, these systems have expanded both in number and complexity and now encompass such diverse application areas as distributed, multimedia and CAD/CAM databases; as well as knowledge bases for AI systems. Hundreds of commercial Data Base Management Systems (DBMSs) are available, targeted for machines ranging from mainframes to personal computers.

Database systems are not without problems however. There are many respects in which current systems need further development: the amount of data that can be stored is often insufficient, although well into the range of trillions of bytes for the largest applications; the response time can be slow, especially for complex transactions like context-sensitive searches or searches of unstructured data such as in full text retrieval systems. The interface to the user is not optimal despite powerful query languages, often of a non-procedural nature. The cost of acquiring and maintaining the hardware-software components of such systems is high, although the performance to cost ratio is being improved continuously.

Database machines, i.e. computers with architectures and software optimized for database management, can help in solving or easing the response/capacity/cost limitations. Evidently, solutions to all of the problems demand much more than an alternative hardware approach. Moving functions from software to hardware and performing as many as possible in parallel, are two directions which lead to performance improvements. Progress in electronic technology, especially with VLSI, has lowered the costs of logic and memory enough to make deviations from the classic general purpose architectures attractive. Unfortunately, the main obstacle, access time for data in magnetic secondary storage has remained essentially constant despite the dramatically increased capacity of the devices themselves.

If one abstracts from the qualities of the proposed or implemented database machines the following are present or desirable: very large storage capacity; use of specialized structures for the disk I/O; memory hierarchy with large data cache; utilization of parallelism and content addressable (associative) memories; special purpose architectures for performing well defined primitive functions like selection, joining or sorting and, finally, operating systems of suitable functionality and performance.

Commercially, the field of database machines is not yet mature, with only a *handful* of products on the market and various research efforts going on at universities and in industrial settings. The issues involved in designing any new architecture are complex and all encompassing, and the traditional ones are so deeply rooted (or well serving) that progress is bound to be rather slow.

With the requirements for database management as given above it is natural to look to optics for possible solutions. This is due primarily to the large storage density achieved in optical disks and the speed and parallelism inherent in light waves. Optical disks, as discussed in the next section, have enormous capacities and although they are currently characterized by various limitations they have the potential of competing successfully with magnetic disks.

The inherent speed and bandwidth of optics has already resulted in major advances in telecommunications because of optical fiber technology. The advantages of optics are beginning to be felt in multiple processor communication and will affect future designs to the interboard, interchip or even to the intrachip levels. The reason is the ability to carry information without interference at GigaHertz rates through guided-wave or free-space propagation.

The development of optical processors is also receiving considerable attention partly because of the high speed of some optical switching elements and partly because of the two-dimensional character of optical processing which suits many problems. All these developments taken together will have significant implications for the design of data and knowledge base systems as we shall discuss in subsequent sections of this article.

Optical storage

Storage is "raw material" for data/knowledge base systems. It is required in great quantity and at the lowest possible cost. Technology has done very well so far in keeping up with (and fueling) the demands; the figures stating the decline of cost per stored bit are always very impressive. In Figure 1 we show rounded values for the most important characteristics of three types of storage/memory devices so that order-of-magnitude comparisons can be made.

	MOS RAM	*Magnetic Disk*	*Optical Disk*
Capacity	1 Mbyte	1 Gbyte	10 Gbyte
Access time	100ns	20ms	100ms
Cost	$100	$10000	$10000
Volatile	Yes	No	No
Erasable	Yes	Yes	No*

Comparison			
Access time	1	$2x10^5$	10^6
Cost/Mbyte ($)	100	10	1

* not presently

Fig. 1 **Order of magnitude figures for storage /memory elements**

The widespread appearance of optical storage can be traced to the introduction of video laser disks a decade ago. In its simplest form, a beam of light detects the presence (or absence) of pits on a revolving reflective layer and servomechanisms are employed for tracking and focusing. The end result is data storage at areal densities an order of magnitude higher than those of the best magnetic hard disks. These properties are even more impressive if we consider the fact that imaging is done from a considerable distance (order of millimeters) and we can dispense with extreme mechanical tolerances and the super clean environment of high performance magnetic disks (Fig 2). This implies cheap removability, an almost ideal solution to the problem of data backup. Another implication is the development of automatic changers or "jukeboxes" with an aggregate capacity in the hundreds of gigabytes.

Magnetic tape economics has a rather different character because, as with removable optical cartridges, the cost of the medium and not of the drives is the most important factor since these systems are primarily used for archival storage. Automated (magnetic tape) cartridge systems are currently available which can hold one Terabyte of data at half the projected cost per Megabyte of optical disks and with an average access time of 10 seconds. Such an access time may seem unacceptable for some applications but one has to be reminded of the 10^5-10^6 ratio in access time that exists between semiconductor memory and magnetic storage. Also, one should be careful in comparing technologies of a radically different age.

	Magnetic	*Optical*
Head/medium gap	0.1 µm	1000 µm
Substrate	Ultra-smooth Al	Al,glass,polymer
Medium thickness	< 1µm	< 1µm
Encapsulation	No	Yes
Environment	Sealed	Open

Fig. 2 Magnetic and optical storage devices

The development of optical disks will almost surely follow the trends present in the well established and mature magnetic disk industry: high performance, state of the art devices from a few manufacturers for the mainframe and advanced applications market, and low-cost, low-end but respectably performing products for a commodity-like market of high volume and intense competition.

Technical and cost limitations have dictated three types of optical storage that can be compared to the ROM, PROM and RAM memories of semiconductor technology. *Read-only* optical disks have their contents fixed at production time, usually by stamping from a master, and will be used mainly for distribution, in a machine-readable form, of materials previously printed or remotely accessed. The strongest contender for the immediate future is the CD-ROM which is a direct descendant of the audio compact disk. It offers a storage capacity of about 600 megabytes on a disk 120mm in diameter (or 5 1/4 " for the recently introduced OROM - Optical Read Only Memory) that can be mastered for approximately $3000 and can be replicated in quantity for $5 each [Chen86]. At least a dozen companies have models either in inventory or production. Prices for the drives range from $500 to $2000 and are bound to decline with massive demand and availability. Having randomly accessible information at the volume and cost that is provided by CD-ROMs opens up a new world of applications. Both professional and consumer markets can be well served and CD-ROMs may well prove to be the second coming of the home computer.

Write-once disks avoid the mastering and stamping steps when mass replication is not needed. Information can be recorded in a non-reversible way which makes this type of storage ideal for archival purposes. But even for operations that do not have an archival nature the extremely large capacity of the disk renders the non-erasability relatively unimportant. Their widespread acceptance is of course subject to the development of operating system support and standardization. Currently available optical disks are of this and the previous type with the largest and more expensive ones tending to be of the write once (or WORM: write once - read many times) type. There is little agreement on the mechanical or electrical characteristics and one of the few common features is the ability to interface with the IBM PC. In the near future it is likely that 10 Gigabyte drives will be available for about $10000, a substantial saving over magnetic disk drives.

Finally, *read/write* erasable disks based on magneto-optic, phase change or mechanical deformation phenomena in a variety of materials may prove to be a superior counterpart to current, state-of-the-art, magnetic disks. For example, at RCA's Advanced Technology Laboratories the Optical Disk Buffer is under development with twelve, double-sided 14-inch erasable magneto-optic disks [Altman86]. The projected performance and comparison with IBM's 3380 is given in Fig. 3. The most important difference is the 200 Mbyte/sec transfer rate which is achieved by 9-track, parallel, spiral recording.

IBM 3380	*Optical Disk Buffer*
30 surfaces	24 surfaces
1.34 x 10^{9} bits/surface	4 x 10^{10} bits/surface
3 Mbytes/sec	200Mbytes/sec
25ms access time	100ms access time

Fig. 3 **RCA's Optical Disk Buffer compared with a high performance contemporary magnetic disk drive**

Another exciting potential made possible by the large medium to head spacing is the rapid deflection of the read/write laser beam between two tracks in less than 100 µs. This will overcome the problem of the slow access time which tends to be the major limitation especially when disk capacities are increased so dramatically. It could be achieved by the variable deflection angle of a grating that can be established by two control beams on a non-linear optical material [Neff87].

Optical storage will most likely complement but not replace magnetic storage. The capability of erasing in real time without undue fatigue is the major technical problem which must be solved before the impact is widely felt. Of course, standardization can neither be hurried nor indefinitely postponed without causing either a blocking of innovation or yet another interfacing nightmare. Software support for new media takes time to develop. Compatibility among the three different types of optical disks is desirable but problematic. But even by conservative estimates the cost of having information available within the access time of a disk will drop dramatically. The natural outcome is to see applications greatly increasing their demands for storage.

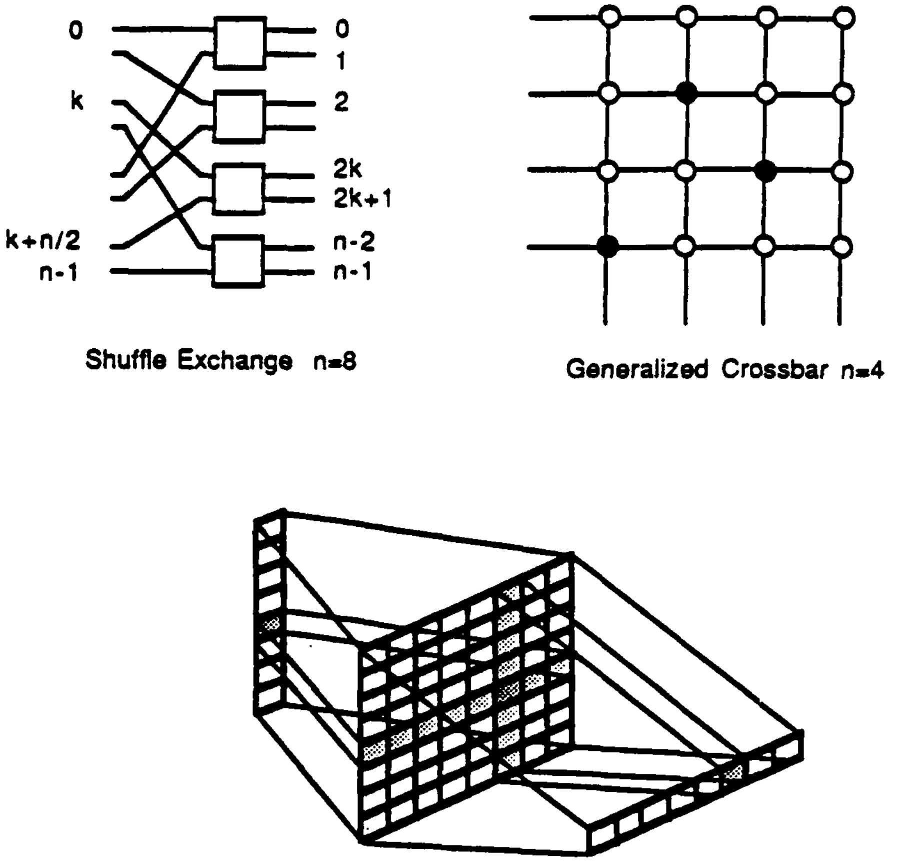
Because of the increased demands, these improvements in storage technology will make corresponding advances in computational performance even more worthwhile. And this will remain true whether or not the technology used is electronic.

Optical Communications

The quest for ever increasing throughput in digital systems has made clock period and pulse widths shrink down to the nanoseconds region. The *bandwidth* needed for transmitting these signals intelligibly is very large. As a result interconnections have become the most critical limitation; for instance in supercomputers we have to use short, expensive to assemble transmission lines instead of the ordinary connection techniques used at lower frequencies. A second problem is *clock skew*, i.e. the difference in arrival time of pulse fronts that originate from distant parts of a circuit. This skew of the input signals could potentially cause a gate to generate erroneous outputs. In order to avoid skew, interconnection length must be restricted and connections with different electrical lengths must be padded to compensate for the different intrinsic delays. To appreciate the problem consider that runs of a few centimeters lead to 1 ns delays which is a significant portion of the clock period. Unfortunately, these problems are not alleviated by the use of VLSI because, as it is widely known, dimension scaling leaves the RC delay time unchanged, at least in a first order approximation.

Currently, the advantages of photons as carriers of information are used to connect machines in local area networks with optical fibers. At a more detailed level, communication between modules of high-speed multiprocessor machines can benefit from either fiber-confined or free- space optical propagation. Further down, chip to chip communication of thousands of signals at the projected high rates could be done optically with less power and interference using *integrated waveguide* optics. Finally, within integrated circuits *reflection holograms* placed at a distance above the plane of the laser diodes and the photodetectors seem particularly well suited for distributing gigahertz clock signals. A comprehensive account of the prospects of optical interconnects can be found in Goodman et al [Goodman84].

Optics may also have solutions for dynamic interconnection needs. We frequently avoid implementing efficient parallel algorithms because of the global, random references they demand. *Crossbar switches* and *shuffle exchange* networks are two topologies that can be beneficially realized with optics. The most important element of the appeal of optics as an interconnect technology is the three-dimensional propagation. For example, an optical crossbar can be very naturally implemented with a column of sources that each illuminate one row of a planar switching array. A row of detectors is placed on the other side with each detector accepting light from a single column of the array (Fig. 4). A 32-by-32 crossbar has been built and larger ones are under development [Bell86].



Shuffle Exchange n=8

Generalized Crossbar n=4

Optical Crossbar n=8

Fig 4 Interconnection primitives and optical implementation of the crossbar

Problems that have to be overcome include the incompatibility of silicon with the materials that are used for fabrication of semiconductor lasers and LEDs, the attachment of optical fibers to integrated circuits, the lack of sensitive holographic materials at the light wavelengths that semiconductors sources produce and the slow response of spatial light modulators. Even with these difficulties though, interconnections may prove to be optics' greatest contribution to computing.

Digital optical processing

The development of the digital computer is so closely associated with the development of electronic technology that we tend to think they are inseparable. But the elementary operations are logical and could be performed by a variety of different technologies. In this section we discuss processing issues in general since it is impossible to separate the functionality required for general purpose computations from that of data / knowledge base processing.

Research in the general area of information processing using optical techniques has been going on for more than three decades. Traditionally, optical processing has been very successfully applied to images and was based on continuous, but often nonlinear, phenomena. Recent advances in optical bistabilities offer promises of a broader view of optical information processing. The possibility of optical logic and optical memory brings digital optical operations closer to reality.

Silicon-based semiconductor logic has already approached the physical limits of switching speed. The investigation of new materials like GaAs has resulted in high speed practical devices which are currently characterized by low integration scale and high cost. Superconducting devices working at cryogenic temperatures (Josephson junctions) looked promising a few years ago, then fell into disadvantage and now are enjoying revived interest because of the recent discovery of high-temperature superconducting phenomena.

Optics may have some answers for these limitations. The large bandwidth, innate parallelism and non-interfering propagation offer mechanisms for overcoming the ever-present communication problems. Switching times for recently demonstrated optical logic elements range from milliseconds to femtoseconds but with an exorbitant amount of energy required for the fastest operations. Although experts agree that it is too early to reach conclusions, there is hope for reducing the necessary energy to a realistic amount

If one is not willing to forgo the flexibility of digital information processing then any viable alternative to semiconductor devices must include:

(a) Elementary switching devices with (usually two, but multivalued logic may also be considered) stable states which can represent information encoded digitally, and

(b) Functional devices, built from the above, such as logic gates and memory cells integrated in dense and low cost packages.

It appears more than one optical phenomenon can be put to work. *Fabry-Perot cavities* and *quantum wells* are two of the structures that have been utilized so far with moderate success.

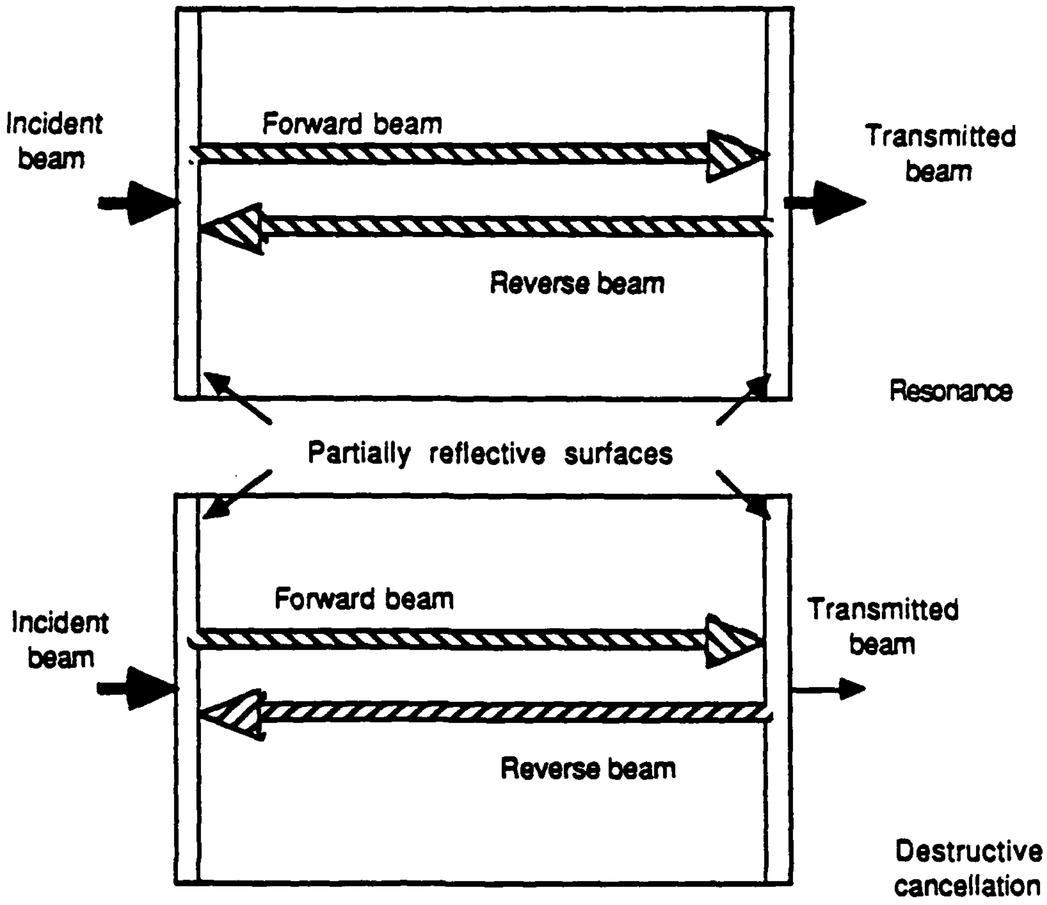


Fig. 5 Fabry-Perot cavity

A Fabry-Perot cavity or etalon (Fig 5) transmits incident light when the optical length of the cavity is an integral multiple of half-wavelengths. If the space between the two partially reflective surfaces consists of an optically nonlinear material (Fig 6) then the transmitted light intensity follows the familiar hysteresis curve (Fig 7).

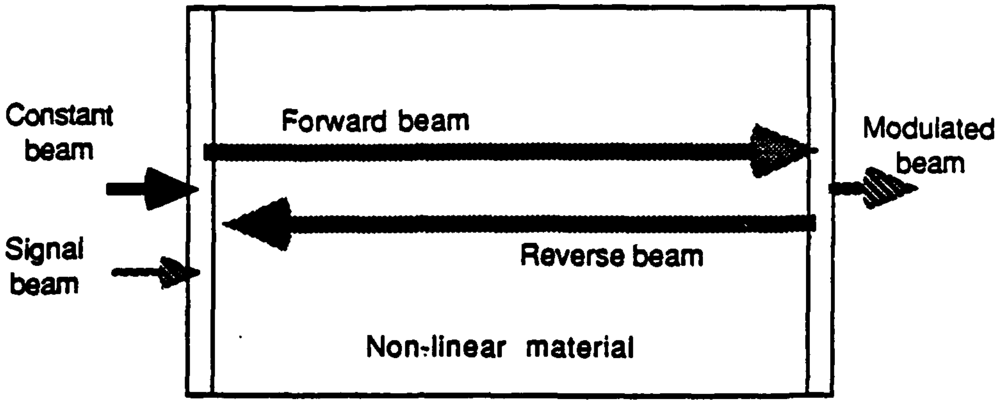


Fig. 6 Optical amplification in a Fabry-Perot cavity

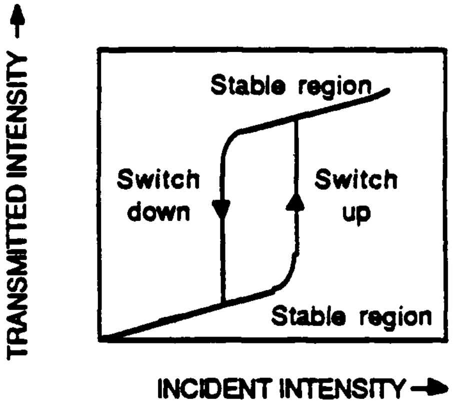


Fig. 7 Effect of a non-linear material in a Fabry-Perot cavity

Quantum-well materials are formed by alternating extremely thin layers of two materials with different electron band gaps. The overall effect is nonlinearities which are far more intense than those that can be obtained using intrinsic semiconductors. Two optical devices based on quantum well phenomena are known as SEED (Self Electrooptical Effect Device) and QWEST (Quantum Well Envelope State Transition) [Bell86].

The above cases must be further researched in order to determine whether they exhibit the required properties of alternative logic elements as given below:

- The input/output transfer function must have a shape similar to the ones shown in Fig 8. The one on the left is the necessary shape for transforming the sum of the intensities of two signals to the AND function and the one on the right is for the OR. For implementing inversion (negation) a descending symmetric curve is necessary.

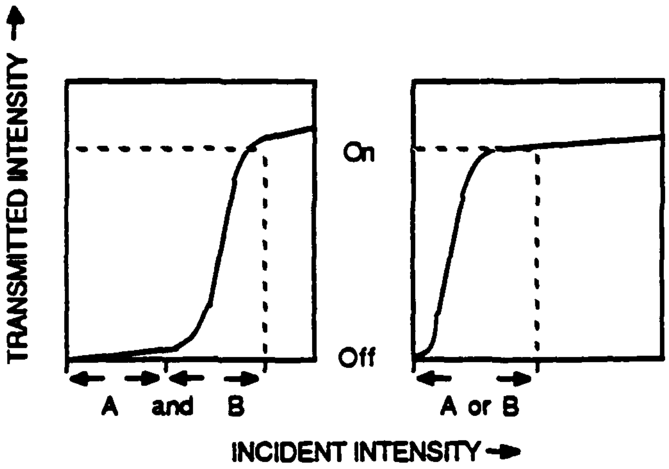


Fig. 8 Implementing Boolean Connectives

- The nonlinearity must have a gain of at least 4 - 10 times to accommodate fanout and distribution losses.

- Logic levels must be restored after each operation because, in contrast with traditional "analog" applications of optics, digital processing requires thousands of operations. This requirement is equivalent to saying that the logic elements must have three "ports", just like an electronic transistor.

- Phase variations (a unique problem due to the wave nature of light) should not affect the function of the logic element. For all possible device states reflections and cavity resonances should be controlled (the equivalent of electrical "impedance matching").

- Though in theory two-input gates are all that is needed, practical systems will be very difficult to design unless four or more mutually non-interacting inputs are provided.
- Tolerance to fabrication variations like doping densities, line widths and surface flatness are essential for low cost as is relative immunity to temperature changes.

The previous account does not exhaust the possibilities for optical logic. *Shadow casting* techniques (albeit relying on optical to electrical conversions) are common and they rely on selective blocking of light by opaque regions. Memory elements are in theory obtainable when amplifying non-linearities are combined in a setup that exhibits positive feedback.

Digital data in most traditional data processing applications, is character-based and so inherently one-dimensional. *Acoustooptic devices* are very good in accurately transforming one-dimensional time-domain signals of extreme bandwidth (order of gigahertz) to spatial perturbations, in effect "freezing" the input signal. Optical processing can then be applied in order to perform common operations like correlation and pattern recognition [Psaltis84]. In the last section of this article we outline the design of an optical comparator that uses multichannel acoustooptic cells to filter high-rate disk data.

For two dimensional data, such as images, *spatial light modulators* are used to modify the intensity of a reference beam of light according to the intensity of an input beam. Operations like amplification, negation and thresholding can be done in parallel for all points on the device surface. Unfortunately the time response of most known devices is currently on the order of milliseconds.

It is too early for a verdict to be reached concerning the prospects of optical techniques in processing, especially if we are to take into account that the yardstick against which they are to be measured is the well matured electronic techniques. Smith and Tomlinson [Smith81] reviewed optical devices and compared them to electronic ones. Although they were reluctant to speak of a digital optical computer they reported superiority of optical devices in the subpicosecond range of switching rates. The switching power required for this ultra fast operation is rather high, with thermal transfer problems as a result, but the authors argued that this is not a very serious obstacle. The reasons are, first, that devices with reactive nonlinearities reflect and do not absorb most of the incident light energy and, second, that the low duty cycle required if these devices are to interface with "slow" electronics greatly reduces the restrictions imposed by thermal dissipation.

The question as to which technology switches the fastest is only part of the overall problem. The communications ability of the technology is at least as important. It becomes evident as architectural questions come into play that an inability to communicate can quickly compromise any advantage a technology might have in terms of speed. To appreciate the severity of this problem one need only consider the rapid degradation of performance in multiprocessor systems when there is appreciable interprocessor communication.

From a simplistic viewpoint, the relative merits of electronics and optics seem to correlate to the properties of electrons and photons. Electrons can influence each other at a distance, so it is easy for an electrical signal to affect another one to perform switching; however, this interaction has unwanted side effects in the form of capacitance and inductance which complicate communications. Photons are the opposite. The absence of interaction gives optics the superior communication qualities but also makes it hard to perform switching. Nevertheless, switching has been achieved at energies comparable to electronics and this may eventually put electronics in a disadvantage.

Architectural issues

The hardware of electronic computers has made tremendous advances in the last 40 years going from vacuum tubes to very large scale integrated circuits. On the contrary, the architecture of computers and the imperative programming languages used to express their programs have remained fundamentally the same because they implement the classic computing model named after John von Neumann. As a result the opportunities offered by VLSI and parallelism are not fully exploited and computer power is reaching the limits imposed by the physical laws of electronics.

The central problem of the von Neumann model is referred to as the *von Neumann bottleneck* and involves the performance limitations that result from the sequential, address-mediated communication between the CPU and memory. It is brought on by the limited number of interconnections that can be supported in a practical manner by an electronics-based technology.

For example, if memory consists of N bytes then it is completely impractical to support N interconnections between the memory and the logic unit with wires (Fig. 9). Using address decoding the interconnections are reduced to $\lceil \log_2 N \rceil$ which is very practical since an immense amount of memory, e.g. 4 Gigabytes, can be addressed with only a few, in this case 32, lines and a prodigious one, e.g. 16 Exa-bytes (18×10^{18}) with only 64. This is the classic space-time tradeoff.

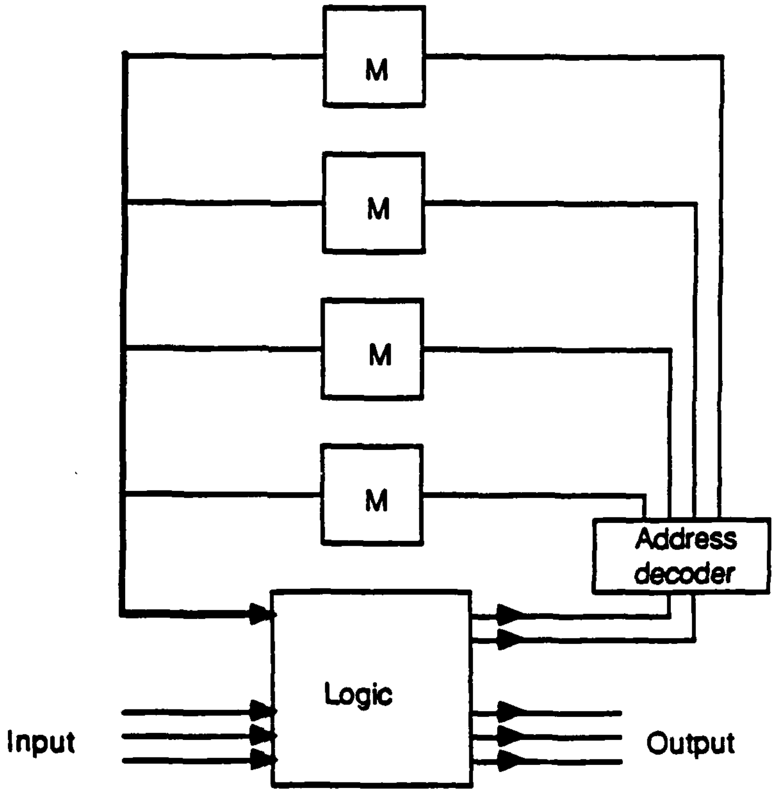


Fig. 9 **Finite State Machine with von Neumann bottleneck**

Constrictions similar to von Neumann's are also found at other levels. At the module level broadcast-bus structures are used for communicating between modules. At the chip level the limited number of external connections forces the use of time multiplexing. In contrast, the number of elements within the chip can be enormous and many applications would benefit from parallel signal communications. .

One of the simplest models for computing, the classic finite state machine does not have this sort of problem. All storage elements are updated in parallel without the need for addresses (Fig. 10). In addition, the storage elements need not preserve their contents for more than one cycle. This may prove valuable in our quest for practical optical memories.

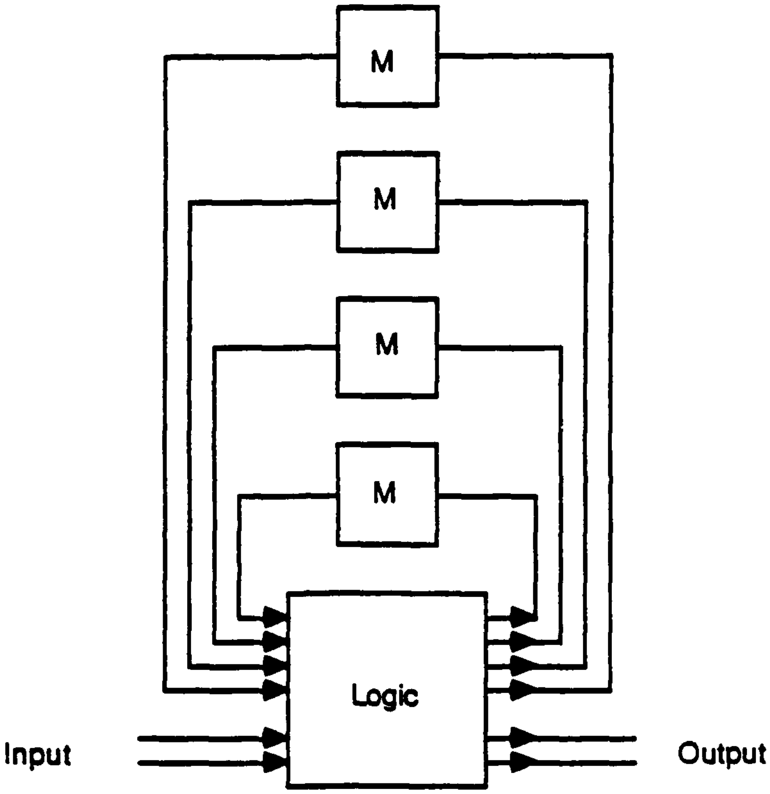


Fig. 10 Classic Finite State Machine

The difficulty with FSMs is that the number of connections between the combinational and memory units is very large. But optical techniques may already have an answer for this. Sawchuck and Strand [Sawchuck84] described an experimental system which was composed of two main components. A spatially parallel array of independent optical gates provided the logic and memory functions, and a computer generated hologram served as a beam-steering element to arbitrarily connect gate outputs to gate inputs in a free space setup (Fig 11). Their system had only 16 gates but it was built to demonstrate the concept by optically implementing a synchronous master-slave flip flop and a driving clock.

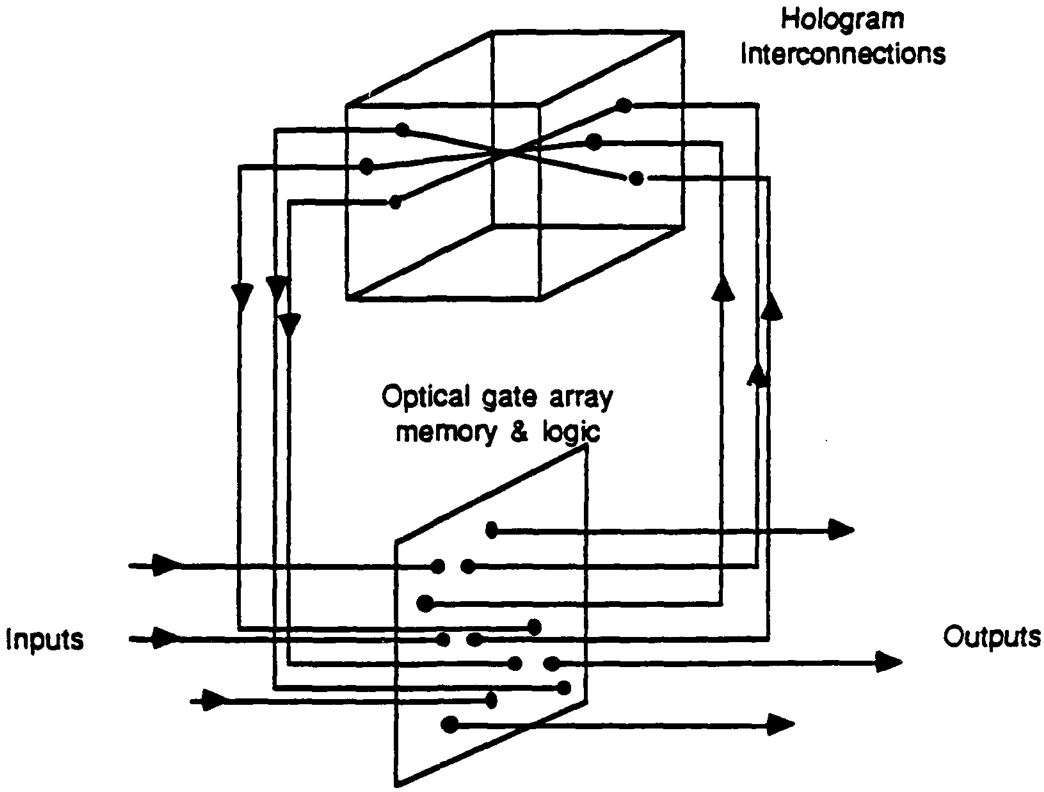


Fig. 11 Free-interconnection non von Neumann processor

It is not clear whether the above results can be extrapolated to systems with thousands or millions of gates . The previous authors state that arbitrarily *space-variant* interconnections are limited by current hologram recording techniques. On the other hand, for *space-invariant* interconnections, like the ones necessary for cellular logic, there is practically no limit even with today's technology.

Another example of an optical processor meant more as an existence proof rather than as a practical design was given by Huang [Huang84]. He demonstrated the practicality of an exceedingly uniform and simple approach based on a functional logic block that can perform all sixteen Boolean connectives such as NOR, NAND, OR etc. These blocks, which are implemented by using only AND gates, are grouped in pairs called *logic cells* so that both a signal and its complement can be simultaneously generated. Finally, pairs of logic cells can be grouped to form a *function/interconnection module* that can be programmed to either perform logic or implement various interconnection primitives. If the latching properties of optical gates (as propagation time approaches switching time) are used for storage and the modules are connected so that the output of the one is the input of the next, then an optical pipelined processor will be formed.

Subsequent research lead to the idea of *symbolic substitution* where the only operation is the recognition of a 2-D subpattern and its substitution by a different one [Brenner 86]. This shows the capability of supporting space-variant operations with a mechanism that is space-invariant. Symbolic substitution has been used to implement Boolean logic, binary arithmetic, cellular logic and simulate Turing machines.

One general observation that should be kept in mind concerning the architecture of optical computers is that an evolutionary approach might not be the best. Optics has unique characteristics and qualities and these should be exploited by appropriate architectures, even if that implies that we have to widen our perception of computing machines. Neural network modeling and other connectionist ideas seem to be one of the alternatives in sight [Mostafa87].

Optics in Data/Knowledge Base machines

The application of optical computing to specific areas should take into account the idiosyncrasies of the problems in the specialized architectures employed in those areas. Some problems may be simplified when a narrower view is taken. In knowledge and data base applications for instance, *selection, projection* and *join* are common processing chores. Search of fixed format data (e.g. indices or pointers) could make effective use of optical content-addressable memory which can be implemented by multiplexing a large number of holograms in a thick recording material like lithium niobate [Gaylord85]. The need for large capacity and high bandwidth secondary storage will probably be satisfied by using optical disks. Optical preprocessing of the retrieved data, without intermediate electrical conversion, will help deal with the extreme data rates.

Currently, access times of optical disks are larger than those of magnetic disks. The reason is that the focusing optics are bulkier than the 'flying' miniature heads of magnetic disks. Data rates are comparable, with potential for improvement since optical disk technology is relatively new . However, in contrast with magnetic media, there are two promising possibilities for increased optical disk performance by at least two orders of magnitude both in terms of access time and sustained data rates. First, as mentioned in the optical storage section, the read/write beam could be deflected from track to track very rapidly (order of 100ìs) by entirely optical means. Second, due to the non-interference of light beams and the relatively large head to medium spacing one could imagine multiple beams being used for reading data with a single head carriage assembly [Carlin84] or an unfocused beam could simultaneously read data from more than one point of a transmissive disk surface [Mostafa87]. This coupled with the possibility of multiple heads will allow for enormous data rates. If one assumes that access times of a 100 µs and data rates of 200 Mbytes per second are achieved then this represents almost two orders of magnitude improvement over current magnetic disks. Input/Output systems will have to be designed with these rates in mind. Current electronics would be hard pressed. However, if data could be preprocessed "on the fly" in its optical form, then the ultimate data rate to the ele.tronics would be much lower on the average, and the data much "richer" in information. Intelligent use of optical pattern matching could provide us with a set of primitive operations that could help implement efficiently higher order functionality like, for instance, a subset of relational algebra operators.

For applications which demand fast searching of many megabytes of data all this is very promising. But with current electronics technology if every subsystem of a machine needs to "cater" to such high rates then its cost will be much higher than necessary. In the remainder of this section we discuss the design of a hybrid opto-electronic preprocessor that can help in this situation. A high-level sketch is depicted in the Figures 12 and 13.

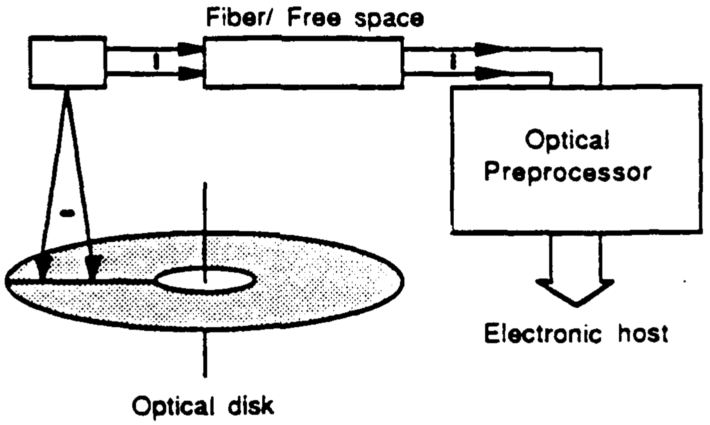


Fig. 12 **Optical communication and processing of high data rate disk output**

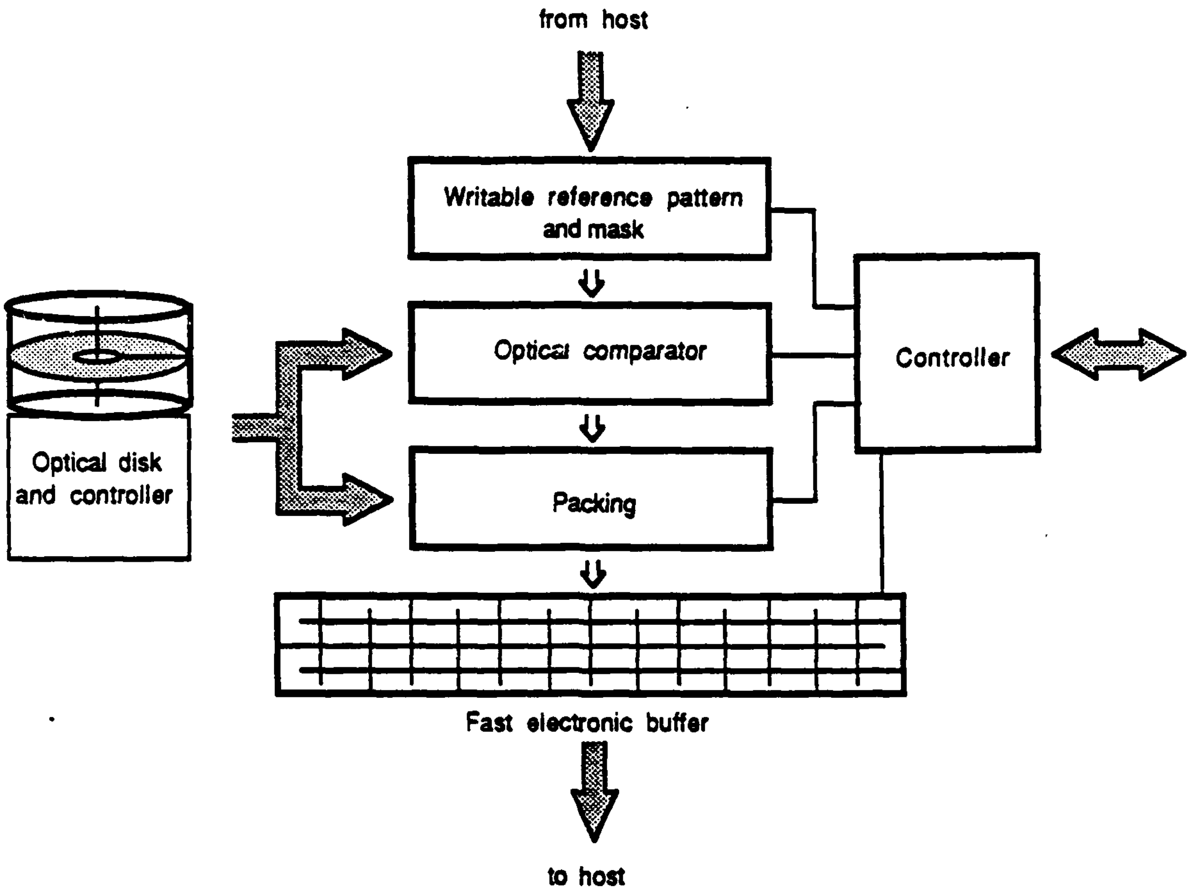


Fig 13 Block diagram of a hybrid opto-electronic preprocessor

The optical comparator receives the error-corrected optical bit stream from the disk which is w-bits wide and it is compared on the fly with an optically-encoded reference pattern. This pattern can contain "don't cares". When the current frame matches the reference pattern the "interesting" portion of the current frame is latched in a large electronic buffer (2-port cache) which holds it until the host is ready to process it. The buffer can be implemented as a ring in order to avoid any internal copying of data. If the buffer ever becomes full then the controller stops the procedure. In this way data rates in the order of 200 Mbytes/sec can be accepted and filtered data can be output on demand at a much lower rate.

The w bits of every symbol are encoded in a dual-rail manner by including also the complement of each bit. Two symbols A and B are equal if

$$A\overline{B} + \overline{A}B = 0$$

in a bit-wise operation. The AND operation is done optically by sequentially propagating a ray of light through two or more points and the OR by imaging two or more rays on the same point [Guilfoyle86].

Figure 14 depicts the flow of data through the process. It consists of the following steps:

1. A w-bits wide stream of data from the optical disk is compared continuously against a reference pattern. The reference pattern may include "Don't cares" which are represented as a pair of zeroes in accordance with the dual-rail encoding. The maximum length that can be matched is n symbols.

2. A match occurs if the OR results are all zero for a length of k symbols, where k is the length of the reference pattern, one of the setup parameters of the buffering operation.

3. A mask specifies which parts of the input stream are of interest and the spatially separated parts are "packed" in order to became contiguous.

4. The packed result is transformed to electric signals and stored in the fast electronic cache before the next match occurs.

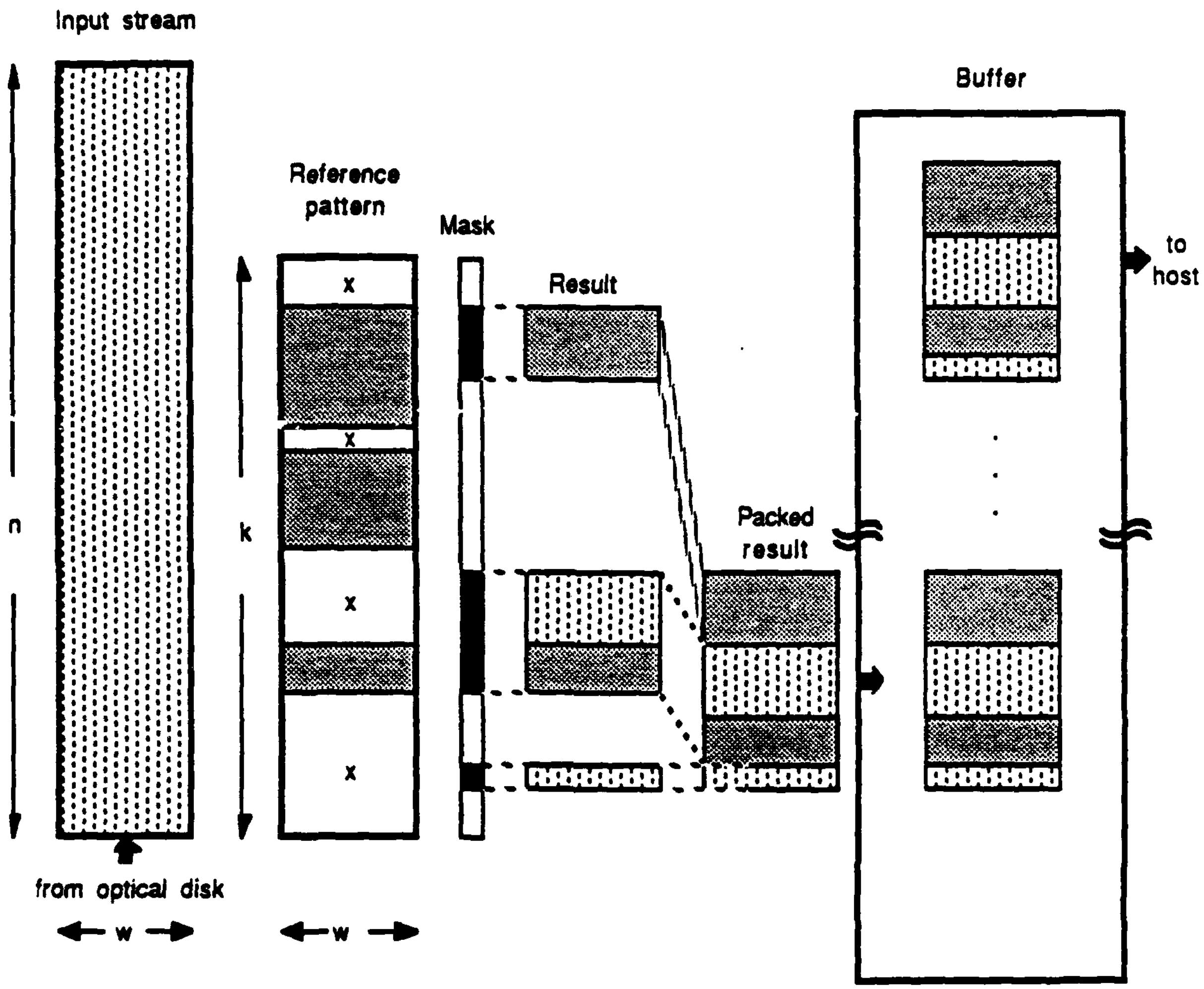
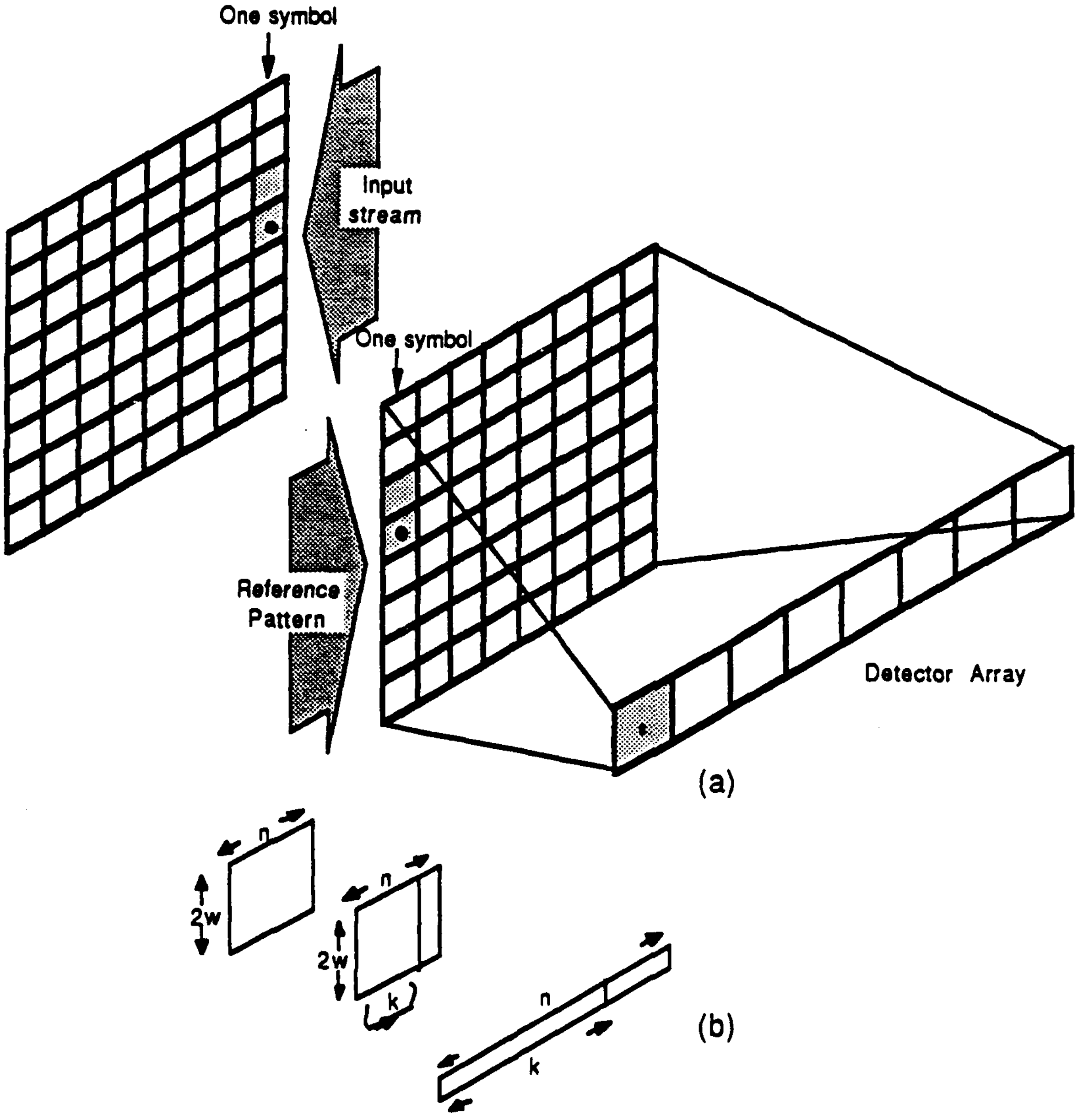


Fig 14 Processing sequence. Input data from the optical disk is filtered and the results are placed in a large, fast electronic buffer

One way that the matching operation can be implemented is shown in Fig. 15. The input data stream and the reference pattern stream enter multi-channel acoustooptic cells from two opposing sides. Light beams are imaged in such a way that the complemented bit, of the input stream symbols are AND-ed with the uncomplemented bits of the reference pattern stream symbols (and vice versa), bit per bit, symbol per symbol. Each detector accepts light from the 2w positions of every symbol (OR operation). When the output is zero on the first k of the detectors then a match has been detected. The operation depends on the circulation of a pattern of length $k \leq n$ symbols in the optical device that is driven by the reference pattern.

When a match is detected the interesting portion of the input pattern according to the mask, is packed and kept in the buffer. Packing entails applying a position-dependent amount of delay to predefined regions of the input pattern while it propagates and hence, should not be very difficult to implement . Finally, the contents of the buffer can be accessed and updated by means of a few, simple electronic counters.



Three dimensional arrangement of the optical symbol matcher.

Fig 15

(a) Each bit of every symbol is represented by two complementary light values that are AND-ed with the corresponding bits of the reference pattern.

(b) The reference pattern is circulated up to a length of k-symbols. A match is detected when the first k detectors register a zero.

In terms of relational algebra operators the preprocessor we have outlined can be employed to perform projection and exact-match selection. In terms of logic-based knowledge bases it can perform filtering of ground clauses. Selection on a conjunction of exact-match criteria is simply accomplished by incorporating all of them in the reference pattern. Disjunction-based selection could be done by using concatenated search patterns if the total length is less than n (and matching on a subset of the detectors) or by connecting more than one optical matcher in a pipeline. Operations that access data repeatedly (like joins) and/or randomly (like sorting) cannot be implemented with a memory-less setup like the one described. Nevertheless, the global connectivity of optics can undoubtedly be exploited with other designs.

At a higher level, the use of electronic content addressable memory has been considered for improving the performance of database operations. Most of these efforts have not met with much success primarily because of the small size and the high cost of these devices and the slow data loading time. On the other hand, optical content addressable memories have the potential for holding megabytes of data at an appreciably lower cost. Since they are hologram-based their major disadvantage is that they are read-only. However, indexing structures to very large data /knowledge bases can be devised which are rather insensitive to updates provided that the update rate is not extreme. Thus, holographic content addressable memories could be used in the future for processing indices to very large data bases and as the field develops they may even be adopted as a primary storage medium.

Conclusion

The issues that are raised by the possibility of a different way of implementing digital computers are far from being just technological. One cannot afford to consider them in isolation and without questioning the applicability of current models for computing. This makes any attempt to pinpoint future directions risky and vulnerable to obsolescence if revolutionary progress takes place. Electronics technology and the von Neumann model have been so well entrenched that straying away from these precepts was not common a few years ago. It also requires competence in a multitude of different scientific fields; it is only recently that we are witnessing such interdisciplinary efforts.

The field of digital optical computing is so young that nearly every kind of research would be beneficial. Optical storage, optical communications and optical devices need development before one can claim that optics has penetrated computer technology. Attacking real problems of a rather limited scope, like the limitations of data / knowledge base machines and pursuing short-term payoffs is the best way to insure sustained interest.

Optical communications and storage are already setting the infrastructure on which more general applications of optics depend. New general directions seem to be worth exploring. Bridging the gap, or sharing ideas between the purely discrete, symbolic world of contemporary computer electronics and the inherently continuous and parallel modeling possible by optical phenomena is one. Dynamic "architecture compilation" driven by the algorithmic requirements of the problem being solved and the resources available is another. In the final analysis optics may have great potential in making computing even more pervasive. Continued efforts are necessary to determine if this potential is real.

References

[Altman86] W.P. Altman, G.M. Claffie, M.L Levene, Optical storage for high performance applications in the late 1980's and beyond, RCA Engineer Magazine, 31-1, January/February 1986

[Bell86] Trudy E. Bell, Optical Computing: A field in flux, IEEE Spectrum, pp 34-57, August 1986

[Brenner86] K.H. Brenner, A. Huang, N. Streibl, Digital optical computing with symbolic substitution, Applied Optics, Vol 25, pp 3054-3060, 1986

[Carlin84] D.B. Carlin, J.P. Bednarz, C.J. Kaiser, J.C. Connolly, M.G. Harvey, Multichannel optical recording using monolithic arrays of diode lasers, Applied Optics, vol 23, no 22, pp 3994-4000, 14 November 1984

[Chen86] Peter Pin-Shan Chen, The compact disk ROM: How it works, IEEE Spectrum, April 1986, pp 44-49

[Gaylord85] T.K. Gaylord, M.M. Mirsalehi, C.C. Guest, Optical digital truth-table look-up processing, Optical Engineering, vol 24, January/February 1985, pp 48-58

[Goodman84] J. Goodman, F. Leonberger. S.Y.Kung, R.A. Athale, Optical interconnections for VLSI systems, Proc. IEEE Vol. 72, July 1984, pp 850-866

[Guilfoyle86] Peter S. Guilfoyle, W. Jackson Wiley, Combinatorial Logic Based Optical Computing, Proceedings SPIE, Vol 639-17, April 1986

[Huang84] Alan Huang, Architectural Considerations Involved in the Design of an Optical Digital Computer, Proc. of the IEEE, Vol. 72, no. 7, July 1984, pp 780-786

[Mostafa87] Yaser S. Abu-Mostafa, Demetri Psaltis, Optical Neural Computers, Sci. Amer., March 1987, pp 88-95

[Neff87] John A. Neff, Major Initiatives for optical computing, Optical Engineering, Vol 26, No 1, January 1987, pp 002-009

[Psaltis84] Demetri Psaltis, Two-Dimensional Optical Processing using One-dimensional Input Devices, Proc. of the IEEE, Vol. 72, Nc. 7, July 1984, pp 962-974

[Sawchuk84] Alexander A. Sawchuk and Timothy C. Strand, Digital Optical Computing, Proc. of the IEEE, Vol. 72, No. 7, July 1984, pp 758-779

[Smith81] Peter W. Smith and W. J. Tomlinson, Bistable Optical Devices promise subpicosecond switching, IEEE Spectrum. Vol. 8 , June 1981, pp 26-33

AN INITIAL DESIGN OF

A VERY LARGE KNOWLEDGE BASE ARCHITECTURE

P. Bruce Berra
Periklis A. Mitkas

Department of Electrical & Computer Engineering
Syracuse University
Syracuse, New York 13244-1240
(315) 423-4445

July 1987

ABSTRACT

In this paper we present an initial design for a Very Large Knowledge Base Architecture. The purpose of this architecture is to serve as a test bed for conducting research in very large data and knowledge bases. When completed the system will contain 100's of gigabytes of magnetic and optical disk storage, at least one half of a gigabyte of solid state memory, parallel paths with very wide bandwidths between elements, and multiple data and knowledge base processors.

This work was supported by the Air Force Systems command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under Contract No. F30602-85-C-0008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC). In addition it was supported by the Defence Advansed Research Projects Agency (DARPA).

AN INITIAL DESIGN OF

A VERY LARGE KNOWLEDGE BASE ARCHITECTURE

INTRODUCTION

Database management, as currently practiced, is a relatively mature field having had its origins in the 1960's. Databases affect our lives whether through weather predictions, airline reservations, stock quotations, food production, football players or unpaid parking tickets. In fact, the control and management of databases is a billion dollar industry that continues to grow and will have an even greater affect on our daily lives in the future. One of the major problems that has plagued us since the origin of the field is how to deal effectively with very large databases. To appreciate the magnitude of some databases one need only think about the hundreds of satellites that are beaming data back to the earth 24 hours/day, seven days a week or the vast information contained in libraries or the information on the millions of tax returns that must be entered into the Internal Revenue Service databases each year.

Partially in response to the problems of large databases, researchers have been very active for the past 20 years in the development of database machines. These machines have as their primary focus the use of parallel processing to improve the performance of the database management function. Secondarily, they are concerned with the implementation of certain database functions in hardware rather than software thus improving performance.

In the last five to ten years there has been considerable interest in the use of database technology to increase the scope of artificial intelligence (AI) applications particularly in the knowledge base area. This has placed new demands on the database field both in terms of increasing functionality and in terms of increased performance.

In order to meet these demands we must develop new knowledge base system architectures that are able to manage very large data and knowledge bases. The new machines must not only be able to process the data from existing databases in the servicing of AI applications such as expert systems, but must also be extended and improved to process and manage knowledge as well. The future for both of these fields is very bright since much of what we now know as AI will be integrated with the database system. To understand this one need only consider the current flurry of activity in the interface between logic programming and relational database, expert database systems and intelligent database systems. This marriage will allow AI researchers to address larger and more complex problems and adds considerable vitality to database systems.

In April 1987 Syracuse University announced the formation of the National Parallel Architecture Center (NPAC) supported by the Defense Advance Research Projects

Agency (DARPA). The initial phase of NPAC concerns the procurement, use and evaluation of innovative parallel computer architectures (i.e. Encore Multimax, Thinking Machines Connection Machine and others). The next phase includes research and development of new parallel computer architectures including the Very Large Knowledge Base Architecture (VLKBA) considered here.

In this paper we discuss the initial design of the VLKBA. The system will have considerable magnetic and optical disk capacity (100's of Gigabytes), wide bandwidth interconnections, large solid state cache memory (1/2 Gigabyte), special data and knowledge base processors and interfaces to several different architectures. The purpose of the system is to serve as a test bed in the study of very large data and knowledge base problems some of which are: query optimization for fast data retrieval, comparison of various knowledge base indexing techniques, interface between Data Base Management System (DBMS) and Logic Programming, back-up and recovery from system failures and real-time applications.

In the first Section of this paper the requirements of a Very Large Knowledge Base (VLKB) are discussed and two architectures are briefly presented that lead to the description of the proposed system in Section 2. The last Section discusses the steps towards building this system and some of its possible applications.

1.1 REQUIREMENTS OF A VERY LARGE KNOWLEDGE BASE

A Knowledge Base (KB) is often defined as a collection of a) facts, b) rules or heuristics about these facts, and c) inferencing mechanisms suitable for reasoning, that allows the system to reach intelligent conclusions. A major problem in the KB design is developing the right techniques and tools for knowledge representation. Four different approaches are dominant in this area namely, Logic Programming, Frames, Semantic Networks and Production Rules, and all of them are based in logic. AI research in the last decade has passed through the development of enhanced AI languages like LISP to the emergence of experimental knowledge base tools like EMYCIN. Now, in the third generation already, fully supported tools for expert systems are commercially available (KEE, Personal Consultant, et al).

Most of these systems are capable of handling Knowledge Bases of only a limited, relatively small size. In the near future, however, Knowledge Bases with data and rules on the order of 10^{11}-10^{13} Bytes and an inference engine that can process hundreds of thousands of rules will be needed so that the next appropriate step is towards architectures for Very Large Knowledge Bases. Obviously conventional techniques are not sufficient for the effective manipulation of such a vast amount of information and new powerful methods are required which will involve extensive parallel processing.

Currently, Knowledge Bases are designed for specific problems such as bacterial infections or nuclear reactor control. As a result their application is limited. In contrast to these homogeneous, narrowly oriented systems, the future, general purpose VLKB will contain different types of information such as: multiple rule sets, many conventional and unconventional data bases, purely numerical data, formatted and unformatted text. This diversion calls for different types of processors too. For example, associative processors, data filters, relational operators, text processors, surrogate file processors etc. The incorporation of all these processing units in the same system along with the efficient integration of Logic and a Data Base Management System will be essential to any future design.

1.2 THE SURROGATE FILE APPROACH

We are investigating various solutions for the management of very large data and knowledge bases in the support of multiple inferencing mechanisms for logic programming. The entire system must operate as a back-end machine removing from the host computer all the time-consuming operations for retrieving and manipulating data.

Since the evaluation of goals can require the accessing of the extensional database (EDB) of facts in very general ways one must often resort to indexing on all fields of the facts. Cast in relational database terminology each relation must be indexed and each attribute of each relation must also be indexed. For very large databases the amount of index data can become very large; in fact it may be as large as the data itself. Thus, if we have 500 Gbytes of fact data we can have 500 Gbytes of index data. We are currently trying to solve this problem [Ber87] with a partial match retrieval mechanism involving surrogate files. These are transformed representations of the index data with most of the information but with only 20% of the size. We have analyzed concatenated code words, superimposed code words and transformed inverted lists as possible structures for the surrogate files.

We will use concatenated code words (CCW) to illustrate some of the ideas. Perhaps the most simplified view of a CCW is that it is just a concatenation of the hashed values of the arguments in a tuple (in binary) with a unique identifier attached to it. The unique identifier is also attached to the tuple and is used as a link (pointer) in the retrieval process. In order to access the database in response to a query from the logic programming inferencing mechanism we must first generate a query code word by transforming the known arguments to a binary representation.

We must place them in their proper location vis-a-vis argument positions in the relation and fill the unknown argument positions with don't care values. This query code word is then used as the search argument in accessing the concatenated code word surrogate file. From this process we obtain a list of unique identifiers that are used to access the extensional database. The retrieved facts are then compared with the original query values to insure that hashing collisions have not occurred and the resulting facts are sent to the logic programming inferencing mechanism.

The use of surrogate files helps to improve retrieval performance because less processing is required due to their smaller size. However, in some cases additional performance can be obtained by distributing the surrogate file entries as uniformly as possible over many disks to allow for parallel processing. We are developing a special Surrogate File Processor (SFP), that will utilize the query code word as a search argument to obtain the list of unique identifiers that qualify. Not only will this surrogate file processor be used for the process discussed above but certain relational operations (i.e. selection, join) can be performed on the CCW thus further improving performance.

The proposed architecture [Ber87] for this system involves several SFPs operating on the disks that contain the surrogate file. The unique identifiers are sent to an extensional data base manager which in turn retrieves the corresponding tuples from the disks containing the EDB.

1.3 THE DATA/KNOWLEDGE BASE PROCESSOR

Instead, shown in Figure 1 is the block diagram of the system where the surrogate file and both the Extensional and Intensional Data Bases are stored in the same group of disks which is controlled by a single Data Collector. Processing is performed by the Data/Knowledge Base Processor (D/KBP) which is directly connected to the host computer.

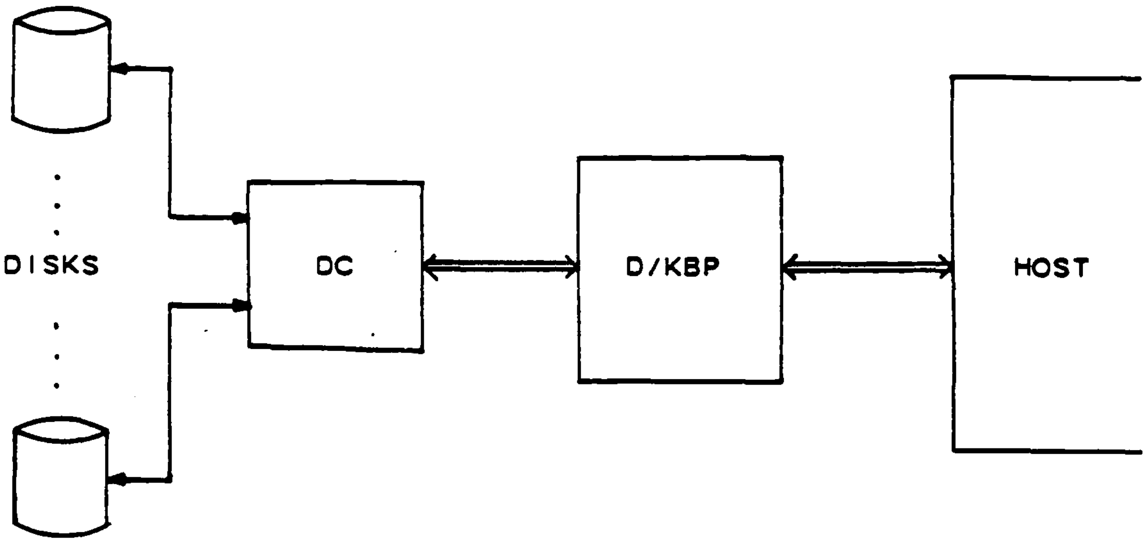


Figure 1. A Data/Knowledge Base Back-End System.

The D/KBP, the heart and the brains of the system, will be a specially designed piece of hardware that processes raw data coming from the disks performing various relational operations, data filtering, sorting, searching etc. It will encapsulate all the processing power necessary to manipulate the Knowledge Base including specialized hardware such as the Surrogate File Processor, sorting pipes, other relational operators and a general purpose processor. Its local memory will be large enough to accommodate the appropriate software for the Inferencing Mechanism and the Data Base Management System as well as the qualified tuples that need further processing.

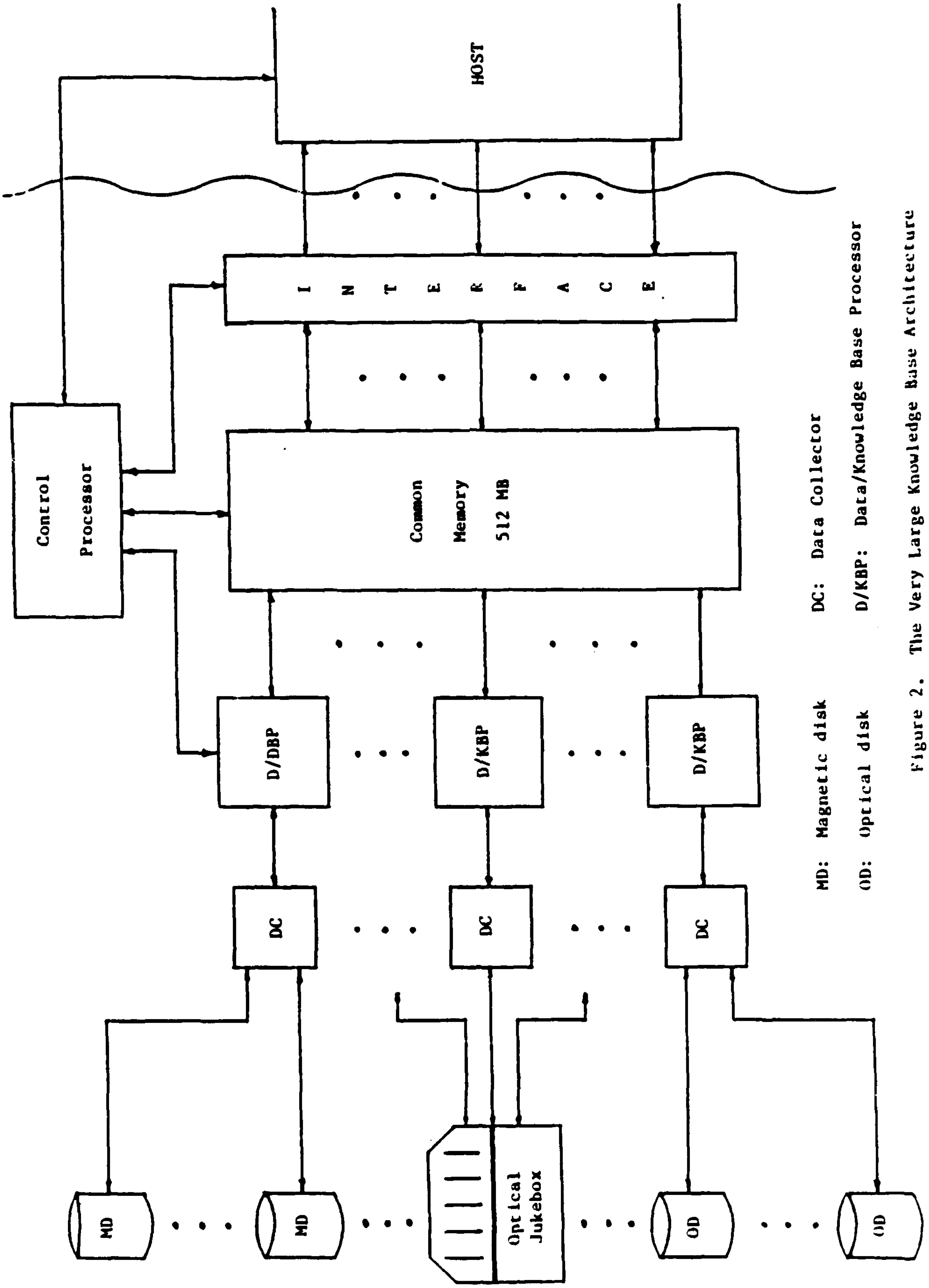
When the host issues a request for a transaction to the D/KBP, the data involved are located on the disks with the help of the Surrogate File Processor, retrieved and placed to the local memory by the general purpose processor. Then a combination of software and hardware techniques are employed in the D/KBP for the efficient data processing so that only useful information is returned to the host.

However, even this configuration is inadequate to handle hundreds of GBytes of data and unable to provide an acceptable I/O transfer rate. We envision a Very Large Knowledge Base Architecture (VLKBA) that will have about 500 GBytes of magnetic and 1500 GBytes of optical disk storage in its full configuration. The VLKBA is the topic of the next section.

2. THE VERY LARGE KNOWLEDGE BASE ARCHITECTURE (VLKBA)

Shown in Figure 2 is an overall diagram of the initial design of a Very Large Knowledge Base Architecture. The VLKBA consists of a large number of secondary storage units (magnetic and optical disks, magnetic tapes) arranged in groups. Each group is controlled by a single data collector which receives data from multiple disks simultaneously and passes them to a Data/Knowledge Base Processor. All the D/KBP's have access to a large semiconductor memory which acts as a disk cache memory between the disks and the host machine. The D/KBPs communicate with each other and with the front-end computer through the common memory. The communication between the common memory and the host is established with an interface that allows for maximum bandwidth and arbitrary channel connection. The entire system is controlled by the Control Processor which accepts requests from the host and translates them to the appropriate commands for the VLKBA.

The primary goal of the design is to achieve maximum performance using a high degree of parallelism. Each part of the VLKBA is described separately in the following paragraphs.



MD: Magnetic disk

OD: Optical disk

DC: Data Collector

D/KBP: Data/Knowledge Base Processor

Figure 2. The Very Large Knowledge Base Architecture

2.1 Memory Storage Units

In order to achieve such a huge capacity we can only consider the largest currently available mass storage media, that is, magnetic disks with movable heads and large optical disks. Some of the most important characteristics of these devices are shown in Table 1.

With the current capacity of the largest magnetic disks being on the order of 5 Gigabytes we need 100 such units to reach the desired 500 GBytes of magnetic storage.

TABLE 1. Secondary Memory Characteristics

	MAGNETIC DISKS (Moving heads)	OPTICAL DISKS (Large-diameter, Write-once)
Capacity (GB)	1 - 5	5 - 10
Transfer rate Burst (MB/sec)	3	0.7 - 10
Transfer rate Sustained (MB/s)	up to 3	0.2 - 1
Average access Time (ms)	15 - 30	150 - 1000
Latency (ms)	8 - 10	20 - 60

In the optical disk area there is a greater variety. Optical disks provide significantly larger capacity but they fall behind in the transfer rate. We are currently examining the possibilities for multiple-beam read from a single disk which could increase the I/O bandwidth dramatically. Another major disadvantage of the optical technology is the inability to change the information once it has been recorded on the optical surface but it seems that this problem will be soon overcome.

The largest part (more than 1000 GBytes) of the optical storage will be provided by a "jukebox" [Amm85, Alt86]; a device that accommodates from 64 to 128 14-inch optical disks arranged in an on-line library configuration and accessed via an automated handling mechanism similar in concept to the well-known music jukebox. For the remaining 500 Gbytes we are planning to use Write-Once Large (14") Optical Disks with a capacity of 10 GB/platter.

A group of disks may be interleaved to speed up data transfers in a manner analogous to the speedup achieved by main memory interleaving. Conventional disks may be used for interleaving by spreading data across disks and by treating multiple disks as if they were a single one. In the synchronized disk interleaving mode [Kim86], every page of the memory is distributed orthogonally over a group of M disks controlled by a common control unit. Every request for a specific page (or a block of more than one page) is broadcasted simultaneously to all M disks that execute the same transaction in parallel. The average service time (ST) for a request is given by:

$$ST = T_a + (T_{tr}/M)$$

where T_a is the average access time (average seek plus average latency time) and T_{tr} is the time to transfer (read or write) a block of data. Path delays due to rotational positioning sensing misses, which are significant in disk systems with skew distribution, are completely eliminated. Obviously, the performance of the design is improved when larger blocks of data are transferred. Synchronized disk interleaving provides simplified control because the interleaved disks can be manipulated as a single unit. Since the load is evenly balanced over all the devices, queueing delays due to multiple requests for a specific disk are avoided, thus allowing for maximum degree of parallelism and considerably lower service time. The reliability of the system can be also improved with minimum redundancy. A typical number for M is 10.

Each D/KBP will have access to 10 synchronized magnetic disks with an overall capacity of 50 GBytes. Every group of disks will be controlled by a separate Data Collector. The Data Collector will receive data from all the disks simultaneously thus obtaining a data transfer rate of about 30 MBytes/sec to each Data/Knowledge Base Processor. Thus, we will have 10 such groups and the total transfer rate can be as high as 300 MBytes/sec. Not shown in Figure 2 is the disk controller. We envision that each disk will have its own control processor and this processor will share the controller function with the data collector.

The average sustained transfer rate from the jukebox is a little below 50 MBytes/sec. Similarly, the low transfer rate from each of the other optical disk drives allows 16 such units to be serviced by a single data collector. Therefore, the output rate from the optical devices will be about 100 MB/sec raising the overall total for the Disks-To-D/KBPs bandwidth up to 400 MB/sec. However, as previously stated, we believe that the data rate from optical disks has the potential to be increased considerably through multi-beam reading. This speculation must await the results of further research.

Provision will be made for at least one magnetic tape unit for back-up purposes.

2.2 The Data/Knowledge Base Processors

The D/KBP accepts data from the data collector and either processes it or passes it through to the common memory. With regard to internal optimization the D/KBP must be able to generate and control index information for the data it manages, it must be able to optimally place the data on disk for minimization of access and update time and it must be able to maintain the data in terms of security, integrity, backup and recovery. Our work with the surrogate files [Ber87] will have a significant impact on the design of the D/KBP.

A more detailed block diagram of the D/KBP is shown in Figure 3. It will contain 8 MBytes of local memory and

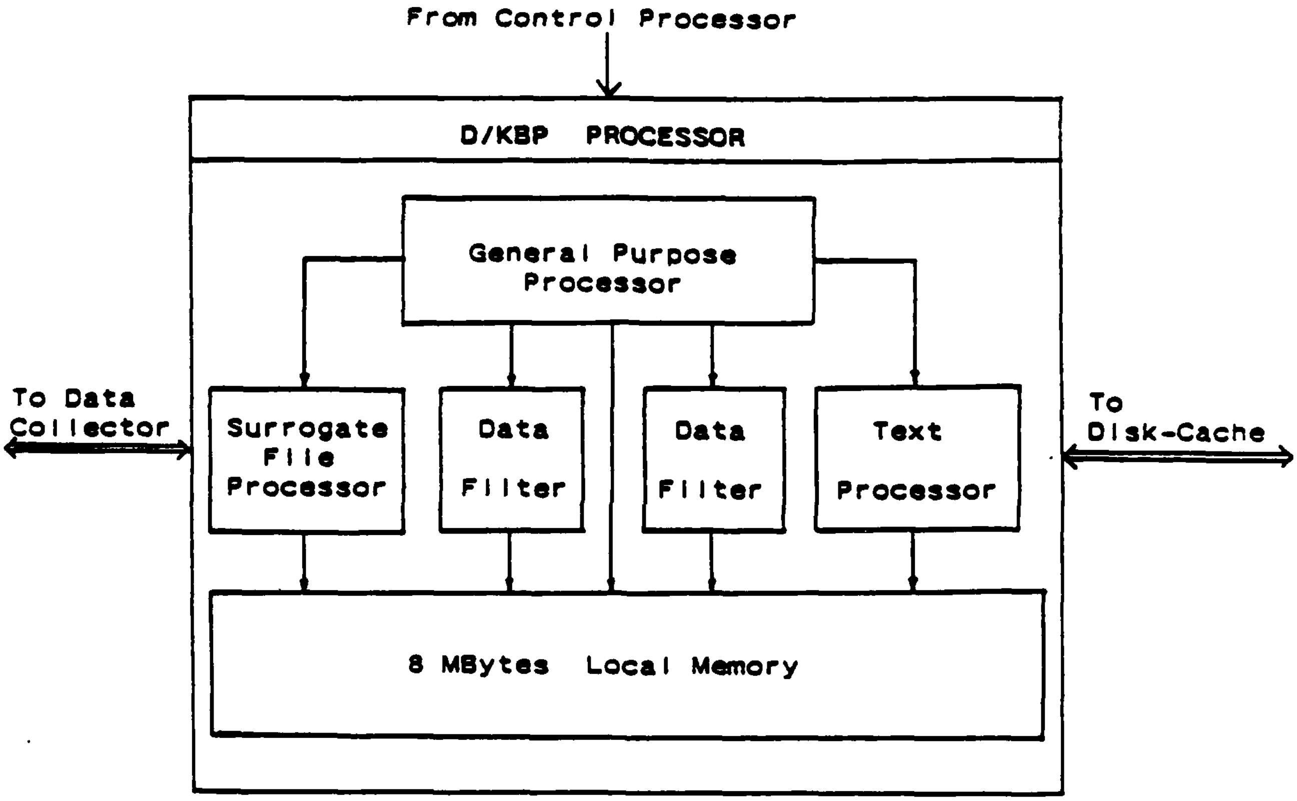


Figure 3. The Data/Knowledge Base Processor.

several specialized processors. The use of the Surrogate File Processor has already been illustrated. The General Purpose Processor will undertake a part of the internal control of the D/KBP and any other job that cannot be performed by the other processors (i.e. numerical computations). In addition the D/KBP contains a filter processor which performs the more common operations such as sort, merge, select, project and join as well as a special text processor.

The D/KBP performs two classes of operations on the data it controls. There are processes that respond to external commands from the control processor as shown in Figure 2 and internal processes that it must undertake to operate in a near optimal way. As previously mentioned we believe that much of the inferencing capabilities of current AI systems will become part of the database system to form an intelligent database or expert database system or perhaps the term knowledge base system will take on that meaning in the future. We believe that new functions will be added to the database system to give it new functionality. For instance in work by Yokota and Itoh [Yoi86] they discuss a relational knowledge base system that has the added functionality of unification-join and unification-restriction. The D/KBP will be designed to include appropriate addition capabilities.

There will be one D/KBP for each group of magnetic disks, three for the entire jukebox (because there are three different channels) and one for every group of optical disks bringing their total number to 16.

Returning to Figure 2 the nonprocessed or processed data are placed in the common memory. These data will be removed via the control processor for some applications but mostly via the interface to the host. The bandwidth between the D/KBP and the common memory and between the common memory and the interface will be on the order of 100's of megabytes so as not to be a bottleneck.

2.3 The Common Memory

The use of a fast electronic buffer, as a disk cache, between the disks and the host offers many advantages; among them, higher bandwidth and synchronous communication. The size of this memory, which may be common to all the D/KBPs, lies in the order of 10^8 Bytes. It must accommodate multiple ports connected in parallel that read and write data simultaneously.

An initial design of such a system consists of the memory partitioned in B banks, where B is the number of the ports connected to it, and the appropriate interconnection network. Each page of the memory is orthogonally distributed over all the banks so that, a word with address (p,d) -- where p is the page number and d the displacement in this page-- is located in the memory bank M(d mod B) and its address in this bank is (pS + d)/B, with S the size of a page in words. Every port i scans continuously the memory banks according to the sequence: i,i+1,...,B-1,0,1,...,i,... Such a distribution physically allows simultaneous access from all ports, even to the same page, without causing any conflicts among them nor any suspension. The access port speed should be equal to that of the host's main memory, or at least half as fast.

This multiport, multiple-access disk cache can significantly enhance the performance of the I/O system. Even if the overall Disk-to-D/KBP bandwidth is less than 400 MB/sec, the transfer rate from the interface to the front-end computer can be considerably higher especially when the disk-cache hit ratio approaches 1.

2.4 The Interface

The interface should allow multiple (on the order of 100) ports from the host computer to be connected to the Common Memory. It will be a perfect shuffle interconnection network. The appropriate connections will be established according to signals from the Control Processor. Generally, more than one host might have access to the Knowledge Base simultaneously.

3. APPLICATIONS AND RESEARCH ISSUES

As pointed out earlier we are concerned with the management of very large data and knowledge bases in a multiple inferencing environment. However, the VLKBA will serve as a resource for many other interesting avenues of research. Among these are questions concerning the management of very large multimedia databases, the development of new embedded architectures, the comparative analysis of database indexing structures, the optimal reorganization of the database in response to usage and some of the more mundane questions concerned with concurrency control, back up, recovery and distribution. In addition, many problems have to be solved in order to achieve the desired cooperation between optical and magnetic equipment.

A promising field for future research is the all-optical data/knowledge base machine. The rapid advance of optical technology, especially in the optical interconnection networks, may soon lead to entirely new architectures for DBMS. Consider, for example, multiple laser beams reading or writing on optical disks at data rates two orders of magnitudes faster than the current ones. This constant stream of data could be guided through optical fibers to an optical computer where many operations could be performed on the data prior to converting it to electronic pulses. Such a system would eliminate the need for data collectors, some of the large semiconductor memory and many of the processing units.

An additional use for VLKBA is the evaluation of experimental machines. When a machine is evaluated on the basis of performance (time per job, throughput, etc.) one must be able to keep the machine supplied with data. In fact, the data rate to the machine under test must be greater than the ability of the machine to handle it in order to obtain a realistic measure of performance. This requirement applies across the entire range of applications from processing intensive applications such as image processing to input/output intensive applications such as data base management. To obtain a realistic measure of a machine's performance, both processing and input/output time must be taken into consideration. Thus, if all of the data will not fit into the machine's main memory, the time to input new data must be taken into account in the performance measure. The time to complete the job then is the sum of the load and process time provided that they cannot be overlapped. In order to ensure a realistic test, the VLKBA must have sufficient capacity and bandwidth to supply the data for a large problem at stress rates to the machine under test.

For problems that are processing intensive, the VLKBA must be able to supply data to the machine being evaluated at the highest rate it can handle. Alternatively, suppose that the machine under test is input/output bound for problems of interest. Then, preprocessing can be performed in the VLKBA in order to enrich the data being sent to the machine under test. Thus, in addition to testing the machine, the VLKBA can identify some of the requirements of the secondary storage system to support the machine being evaluated.

The testing of machines places some severe constraints on the VLKBA. It must be able to sustain output data rates in the hundreds of megabytes per second range and have a data capacity in the hundreds of gigabytes. It must have the facility to provide raw data to a machine under test at stress data rates and must also be able to perform considerable processing activities in order to enrich the data being sent to the machine under test. It must be able to interface with a wide variety of machines. It must have some level of reconfigurability so that the above functions can be performed, and it must be partitionable so that it can interact with more than one machine simultaneously.

CONCLUSION

In previous sections we outlined the architecture of a very large knowledge base system, some of the AI/data base research projects it would support and its use in the evaluation of parallel computers. The design and development of the system is a significant research project in itself and requires a carefully planned phased approach.

The overall design will be such that the system's generality is maintained in terms of reconfigurability, partitionability and intelligence while allowing for gradual growth. We plan to design and build the system as research progresses. Specifically, the VLKBA will gradually evolve from a prototype system through a series of design stages, from essentially a small disk farm to a large intelligent processing system. We are currently developing the surrogate file processor which will be part of and influence the design of the D/KBP, we are conducting research aimed at the integration of AI and data base and we are investigating the feasibility of an all optical database machine. All of these projects will have considerable influence on the VLKBA.

REFERENCES

[Amm85] Ammon, G.J., Calabria, J.A., Thomas, D.T., "A High-Speed, Large-Capacity, "Jukebox" Optical Disk System," IEEE Computer, Vol. 18, pp. 36-45, July 1985.

[Alt86] Altman, W.P., Claffie, G.M., Levene, M.L., "Optical Storage for High Performance Applications in the late 1980s and beyond," RCA Engineer, Vol. 31, Jan./Feb. 1986.

[Ber87] Berra, P.B., Chung S.M., Hachem N.I., "Computer Architecture for the Processing of a Surrogate File to a Very Large Data/Knowledge Base," IEEE Computer, Vol. 20, pp. 25-32, March 1987.

[Kim86] Kim, M., "Synchronized Disk Interleaving," IEEE Transactions on Computers, Vol. C-35, pp. 978-988, November 1986.

[Yol86] Yokota, H., H. Itoh, "A Model and an Architecture for a Relational Knowledge Base," The 13th Annual International Symposium on Computer Architecture, pp. 2-9, June 1986.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.